软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2022,33(8):2755−2768 [doi: 10.13328/j.cnki.jos.006596]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

# 面向 SQLite3 数据库 API 调用序列的并行运行时验证方法*

于 斌[1,2], 陆 旭[1,2], 田 聪[1,2], 段振华[1,2], 张 南[1,2]

[1](西安电子科技大学 计算机科学与技术学院, 陕西 西安 710071)

[2](综合业务网理论与关键技术国家重点实验室(西安电子科技大学), 陕西 西安 710071)

通信作者: 田聪, 段振华, E-mail: {ctian, zhhduan}@mail.xidian.edu.cn

**摘 要**: 作为轻量级的高可靠嵌入式数据库, SQLite3 已被广泛应用于航空航天和操作系统等多个安全攸关领域, 其提供了丰富灵活 API 函数以支持用户快速实现项目构建. 然而, 不正确的 API 函数调用序列会导致严重后果, 包括运行错误、内存泄露和程序崩溃等. 为了高效准确地监控 SQLite3 数据库 API 函数的正确调用情况, 提出了基于多核系统的并行运行时验证方法. 该方法首先分析 API 函数文档, 自动挖掘相关 API 调用序列规约描述, 辅助人工将其形式化表达为具有完全正则表达能力的命题投影时序逻辑公式; 然后, 在程序运行时, 采用多任务调度策略, 将程序执行产生的状态序列分割并对不同片段并行验证. 实验结果表明: 该方法能够发现调用 SQLite3 数据库 API 函数的 30 个被验证 C 程序中, 违背 API 函数调用序列规约的达 16 个. 另外, 与传统串行运行时验证方法的对比实验表明, 提出的并行运行时验证方法能够有效提高多核系统的验证效率.

**关键词**: SQLite3; API 调用序列; 命题投影时序逻辑; 并行; 运行时验证
**中图法分类号**: TP311

中文引用格式: 于斌, 陆旭, 田聪, 段振华, 张南. 面向 SQLite3 数据库 API 调用序列的并行运行时验证方法. 软件学报, 2022, 33(8): 2755–2768. http://www.jos.org.cn/1000-9825/6596.htm

英文引用格式: Yu B, Lu X, Tian C, Duan ZH, Zhang N. Parallel Runtime Verification for Calling Sequences of SQLite3 Database APIs. Ruan Jian Xue Bao/Journal of Software, 2022, 33(8): 2755–2768 (in Chinese). http://www.jos.org.cn/1000-9825/6596.htm

## Parallel Runtime Verification for Calling Sequences of SQLite3 Database APIs

YU Bin[1,2], LU Xu[1,2], TIAN Cong[1,2], DUAN Zhen-Hua[1,2], ZHANG Nan[1,2]

[1](School of Computer Science and Technology, Xidian University, Xi’an 710071, China)

[2](State Key Laboratory of Integrated Service Networks (Xidian University), Xi’an 710071, China)

**Abstract**: As a lightweight and highly reliable embedded database, SQLite3 has been widely used in many security-critical areas such as aerospace and operating systems. It provides rich and flexible API functions to support users to quickly construct projects. However, an incorrect API function call sequence can cause serious consequences, including runtime errors, memory leaks, orprogram crashes. In order to efficiently and accurately monitor the correct call of SQLite3 database API functions, this paper presents a parallel runtime verification approachfor multi-core machines. This approach first analyzes API function documents, automatically mines API call specification descriptions, and assists humans to formalize them as propositional projection temporal logic formulaswith the full regular expressiveness. Then, while the program is running, the multi-task scheduling strategy is employed to divide the generatedstate sequence into several segments and achieve the parallel verification for different segments. Experimental results show that the proposed approachis able to find that among the 30 programs invoking SQLite3 database API functions, there are 16 violations of the API call sequence specifications, with a violation rate of 53%. In addition, with comparative experiments of traditional sequential runtime verification approaches,it is shown that the proposed parallel runtime verification in this study can effectively improve the verification efficiency in a multi-core system.

* 基金项目: 国家重点研发计划(2018AAA0103202); 国家自然科学基金(61732013, 62172322, 62002290); 中央高校基本科研业务费专项基金(XJS210305); 陕西省自然科学基础研究计划(2021JQ-208)

本文由“形式化方法与应用”专题特约编辑陈立前副教授、孙猛教授推荐.

收稿时间: 2021-06-26; 修改时间: 2021-10-14; 采用时间: 2022-01-10; jos 在线出版时间: 2022-01-28

**Key words**: SQLite3; API call sequence; propositional projection temporal logic; parallel; runtime verification

SQLite3 作为开源的嵌入式数据库，已被广泛应用于航天软件、操作系统和反病毒软件等多个安全攸关领域，具有体积小、速度快和可靠性高等特点. Firefox 和 Facebook 等网络浏览器和知名网站均使用 SQLite3 管理日志数据[1]. 为方便用户实现特有的定制化功能，SQLite3 提供了 200 余个 C 程序应用程序接口(application programming interface, API)[2]. 这些 API 函数的调用能够实现对数据库的特定操作，例如，分别通过调用 *sqlite3_open* 和 *sqlite3_close* 函数打开和关闭数据库.

API 函数的灵活调用，使得用户能够快速构建项目. 然而，此过程需要用户花费大量时间去学习相应 API 函数的正确使用方法，从而对每个被调用函数具有深刻理解. 研究表明，软件开发人员需花费整个开发流程的 40%时间成本用于学习 API 函数[3]. 其中，除了每个 API 函数的实现功能以及调用方法保证了单个 API 函数的正确调用外，多数情况下，若干 API 函数需按照指定的调用序列共同完成一项任务[4]. 虽然在 API 函数的文档描述中，这些调用序列规约大多会用更加明显的形式加以强调，然而对于 API 函数的调用者而言，在实际使用中，完全符合全部要求仍面临巨大挑战，从而导致内存泄露、运行错误和程序崩溃等严重后果.

尽管多个领域的 API 函数调用序列已能够通过静态或动态方法进行验证，例如 Java 软件应用[5−8]、硬件基础设备[9,10]和网络服务协议[11−13]，但在验证 SQLite3 数据库 API 函数的调用是否符合调用序列规约方面，仍缺少相关研究工作. 此外，现有研究工作更为看重能够发现的规约违反数量，在一定程度上忽略了验证效率.

作为一种新型的轻量级形式化验证技术，运行时验证[14]将传统的形式化验证与程序动态运行相结合，能够监控待验证系统的实际运行过程，一旦系统的行为违反给定的性质，及时作出反应或发出警告，从而对检查出的错误给予干预和调整，保障待验证系统的正确性. 运行时验证技术已在多个领域得到较为成熟应用，如针对 Java 程序运行时行为监控[15]、普适计算应用时空性质验证[16]和数据流时序性质检测[17]等. 在运行时验证中，时序逻辑语言常被用来描述我们期望该系统满足的性质，大多运行时验证工具使用线性时序逻辑(linear time temporal logic, LTL)或计算树逻辑(computation tree logic, CTL)来形式化描述性质[18,19]. 然而，由于这些逻辑语言无法直接表达完全正则性质，因此他们的表达能力是有限的. 具体来讲, LTL 和 CTL 不能直接表达两类性质：(1) 与时间相关的性质，例如“函数 $f$ 在第 100 个状态之后并且在第 200 个状态之前被调用”；(2) 周期重复的性质，例如“某个函数每隔 100 个状态重复被调用”. 此外，大多数现有的运行时验证方法只是采用串行方式对程序执行产生的状态序列进行顺序验证[20,21]，在这些验证方法中，多核机器的计算资源往往得不到充分利用. 由于验证的过程中需对程序产生的每个状态进行分析，其速度远远小于程序的执行速度，因此，采用顺序的串行验证方式会导致程序运行过程中，问题的发现大幅度晚于问题的发生，从而延误错误修改.

基于以上分析，本文提出了针对 SQLite3 数据库 API 调用序列的并行运行时验证方法. 首先，提出了关键信息过滤框架，对 SQLite3 官方网站上公布的 API 调用序列规约进行挖掘，并利用具备完全正则表达能力的命题投影时序逻辑(propositional projection temporal logic, PPTL)[22,23]对其形式化表达；进而，基于待验证的 API 调用序列规约库，利用底层虚拟机(low level virtual machine, LLVM)[24]平台，对调用 SQLite3 相关 API 的 C 程序进行插桩；最后，使用运行时验证方法监控程序的执行. 在此过程中，为了提高验证速度以便及时发现规约违反问题，我们将程序运行产生的状态序列分为若干片段，为每一个序列片段创建一个验证线程，以使得这些片段能够被并行验证. 当这些序列片段验证完成后，将各个片段的验证结果进行合并，得到最终的验证结果. 在对 30 个程序进行验证后发现，违反 API 调用序列规约的比率达到 53%. 此外，在效率提升方面，本文提出的并行运行时验证方法能够有效发挥多核机器硬件能力，提高验证效率.

本文第 1 节介绍 SQLite3 数据库和 API 函数的基本概念、PPTL 的语法语义以及传统的运行时验证方法. 第 2 节介绍常见的两类违反 SQLite3 数据库 API 调用序列规约的程序实例. 第 3 节给出 API 调用序列规约的挖掘方法以及规约库的构建. 第 4 节详细描述基于多核系统的并行运行时验证方法的框架和实现方法. 第 5 节展示实验结果，并对实验结果进行分析. 第 6 节给出相关工作. 第 7 节总结全文并展望未来工作.

# 1 基础概念

## 1.1 SQLite3数据库及API函数介绍

作为一个轻量级的关系数据库, SQLite3 在使用前无需提前配置系统运行环境, 能够高效利用嵌入式系统有限的资源实现事务处理功能, 具有系统开销小和实时性能好等优势. SQLite3 数据库文件是一个单独的普通磁盘文件, 能够被定位在系统路径的任意位置, 只要 SQLite3 具有读写磁盘文件的权限, 就能直接访问数据库文件.

SQLite3 采用模块化设计, 主要包括内核模块、SQL 编译器模块、后端模块以及附件共 4 个部分, 内部结构如图 1 所示.

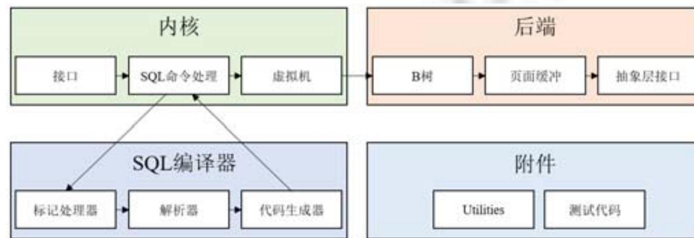


图 1 SQLite3 架构图

作为已经编写好的 C 程序子模块, SQLite3 的 API 函数能够被用户直接调用, 从而使得 SQL 语句被 SQL 编译器高效处理. 具体来讲, SQL 语句首先被标记处理器分解成标识符, 进而交由解析器对其进行识别, 在标识符被重新组合后, 由代码生成器生成相对应的虚拟机器码. 虚拟机作为 SQLite3 内部结构的核心部分, 负责与数据相关的全部操作, 实现用户和存储间的信息交换. 通过执行虚拟机器码, 虚拟机最终实现 SQL 语句指定的功能. 系统硬盘上的数据库文件按照 B 树形式进行存储, 利用页面缓冲机制快速存储和查找数据. 在数据库底层, SQLite3 采用抽象层接口实现不同系统的移植操作. 除此之外, SQLite3 还包括 Utilities 模块中的内存分配与字符串比较等程序以及针对数据库代码的相关测试模块.

我们以最常使用的数据库文件打开和关闭函数为例, 介绍 SQLite3 的 API 函数格式与基本用法.

(1) *int sqlite3_open(constchar*filename,sqlite3**ppDb)*

打开指定的数据库, 其中, 数据库文件路径和名称由参数 *flilename* 指定. 若此路径下存在文件, 则打开该文件; 若不存在, 则创建相应的数据库文件. 针对以上两种情况, 参数 *ppDb* 返回相应的数据库句柄, 同时, 函数的返回值为 SQLITE_OK; 否则, 函数的返回值为对应的异常代码.

(2) *int sqlite3_close(sqlite3*db)*

关闭已经用 *sqlite3_open*(·)函数打开的数据库句柄为 *db* 的数据库. 需要指出的是: 即使数据库文件未被 *sqlite3_open*(·)函数成功打开, *sqlite3_close*(·)函数仍需被调用来释放已分配的资源. 当数据库文件被 *sqlite3_close*(·)函数成功关闭时, 函数返回值为 SQLITE_OK; 在关闭失败情况下, 返回值为 SQLITE_ERROR; 在数据库尚未完成全部相关操作情况下, 返回值为 SQLITE_BUSY.

## 1.2 命题投影时序逻辑PPTL

PPTL 公式 $P$ 归纳定义如下[22,23]:

$$P::=p|\bigcirc P|\neg P|P_1\vee P_2|(P_1,\ldots,P_m)\ prj\ P|P^+,$$

其中, $p$ 为可数原子命题集合 $Prop$ 中的一个原子命题, $P_1,\ldots,P_m$ 和 $P$ 是 PPTL 公式. 状态 $s$ 为从 $Prop\rightarrow B=\{\text{true}, \text{false}\}$的映射, 在状态 $s$ 上, 原子命题 $p$ 的取值用 $s[p]$表示. 区间$\sigma=\langle s_0,s_1,\ldots,s_i\rangle$表示状态的非空序列, 其长度可为有穷或无穷. 令$|\sigma|$表示区间$\sigma$的长度, 若$\sigma$为有穷的, 则$|\sigma|$为区间上的状态数减 1; 若$\sigma$为无穷的, 则$|\sigma|=\omega$. 关系符号$<<$定义为$\leqslant\backslash\{\omega,\omega\}$, 表示从关系操作符$\leqslant$中除去$\omega=\omega$. $\sigma_{(i..j)}$表示区间$\sigma$上的子区间$\langle s_i,\ldots,s_j\rangle$, 两个小区

间可以通过符号·连接成为一个大区间. $\sigma$在 $r_1,\ldots,r_h$ 上的投影被表示为 $\sigma\downarrow(r_1,\ldots,r_h)=\langle s_{t_1},\ldots,s_{t_l}\rangle$, 其中, $t_1,\ldots,t_l$ 是 $r_1,\ldots,r_h$ 上最长的严格递增子序列.

PPTL 的语义以及面向 PPTL 性质的传统运行时验证方法已在文献[25,26]中给出, 在此不再赘述.

## 2 实例分析

本节将展示 C 程序调用 SQLite3 数据库 API 函数的过程中, 两种经常出现的不符合调用序列规约情形[25].

### 2.1 调用废弃的API函数

随着软件实现功能的不断演化, 一些早期的 API 函数已不能完全符合安全或者性能等方面的要求, 此时会存在对应的新版本实现并优先推荐被调用. 然而, 由于兼容性方面的考虑, 避免导致调用早期 API 函数的相关程序发生崩溃, 不能立刻取消对早期API函数的支持. 在此情况下, 这些API函数需要首先被提供者标记为废弃的 API 函数, 用以提示使用者在新版本中尽快取消或修改相关调用. 由于随时可能被停止支持, 因此应避免现阶段对废弃 API 函数的调用.

下面我们以 *fast_vacuum* 程序为例, 展示此情况实例. 该程序是 SQLite3 数据库源代码中的自测程序, 实现对数据库的重新布局, 顺序排列文件内容以尽可能多地消除尚未使用的空间, 高速完成 VACUUM 命令. 其中, 以数据库文件句柄 *db* 和 SQL 语句 *zSql* 为参数, 图 2 所示的 *execSql* 函数被多次调用以执行 SQL 语句指令. 在此函数中, 针对 SQL 语句对象主要有 3 步操作: (1) 通过调用 *sqlite3_prepare* 函数完成准备工作; (2) 通过调用 *sqlite3_step* 函数运行指定 SQL 语句; (3) 通过调用 *sqlite3_finalize* 函数释放 SQL 语句对象.

需要指出的是: 上述被调用的 *sqlite3_prepare* 函数是废弃 API 函数, 其无法在 SQLite3 数据库发生意外时进行必要的回滚操作, 目前仅是为了兼容性而暂时保留. 为了弥补此缺陷, SQLite3 发布了用以替换 *sqlite3_prepare* 的新 API 函数 *sqlite3_prepare_v2*. 针对这一点, *sqlite3_prepare_v2* 函数的使用描述中专门指出: "函数 *sqlite3_prepare* 是遗留 API 函数, 应避免继续使用". 除此之外, *sqlite3_step* 和 *sqlite3_finalize* 函数的说明中还分别提到: "在一条 *prepare* 语句完成后, 必须调用 *sqlite3_step* 函数执行该语句"和"为了避免泄露相关资源, 程序必须结束每一条 *prepare* 语句".

基于以上分析可以看出, 3 个 API 函数 *sqlite3_prepare_v2*, *sqlite3_step* 和 *sqlite3_finalize* 必须被顺序地循环调用. 此调用规约为一个用 LTL 和 CTL 公式难以直接表达的完全正则性质, 但是可用以下 PPTL 公式直接表达:

$$\Diamond((sqlite3_prepare_v2=1;sqlite3_step=1;sqlite3_finalize=1)^{+}).$$

公式中, 变量等于 1 表示程序中同名 API 函数被调用. 此公式的含义为: 一个周期重复的性质从某时刻起成立, 其重复次数大于 0; 且在每个周期内, 函数 *sqlite3_prepare_v2*, *sqlite3_step* 和 *sqlite3_finalize* 被依次顺序调用.

```
1  void execSql(sqlite3 *db, const char *zSql){
2    sqlite3_stmt *pStmt;
3    if(!zSql){
4      fprintf(stderr, "out of memory!\n");
5      exit(1);
6    }
7    printf("%s;\n", zSql);
8    if(SQLITE_OK!=sqlite3_prepare(db, zSql, -1, &pStmt, 0)){
9      fprintf(stderr, "Error: %s\n", sqlite3_errmsg(db));
10     exit(1);
11   }
12   sqlite3_step(pStmt);
13   int rc=sqlite3_finalize(pStmt);
14   if(rc){
15     fprintf(stderr, "finalize error: %s\n", sqlite3_errmsg(db));
16     exit(1);
17   }
18 }
```

图 2 程序 *fast_vacuum* 中的 *execSql* 函数

### 2.2 缺失必要的API函数

缺失必要的 API 函数是指在使用一个或多个 API 函数实现某功能后, 未进一步调用指定 API 函数完成后续操作. 我们以 GitHub 上公开的 C 程序 *invertory* 为例展示此情况. 在该程序中, 商品货物的必要信息被储存在一个 SQLite3 数据库文件中, 货物信息的插入、删除和查询等功能将通过调用有关 API 函数来实现. 以插入功能为例, 不同模块之间的调用关系如图 3 所示.

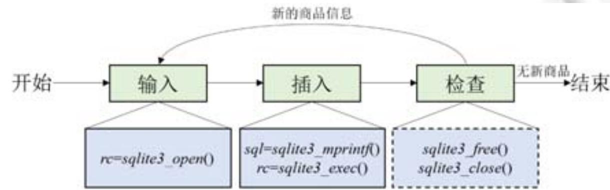


图 3 *invertory* 程序中完成商品信息插入功能所需调用的模块

图 3 中, 前两个模块下方的实线矩形框包含了模块所调用的 API 函数, 最后一个模块下方的虚线矩形框展示了缺失调用的 API 函数. 商品信息的插入过程可以分为如下 4 个步骤: (1) 打开指定数据库文件; (2) 字符串 *sql* 作为 SQL 语句, 以特定形式保存商品名称与价格; (3) 以 SQL 语句 *sql* 作为参数, 将商品信息插入到数据库中; (4) 如还需将其他商品信息添加到数据库中, 重复上述过程; 否则, 停止此过程. 图 4 代码块中的 *insertItem* 函数实现了上述操作, 函数参数 *dbname* 为保存商品信息的数据库名称, 变量 *db* 和 *sql* 分别为数据库句柄和保存 SQL 语句的字符串, 变量 *rc* 用于保存不同 API 函数被调用后的返回值.

在 *insertItem* 函数中, 首先通过调用 API 函数 *sqlite*3_*open* 实现数据库文件创建, 分配与数据库句柄相关的资源. 值得注意的是: 成功分配的资源不会自动释放, 当不再需要此数据库句柄时, 需显式调用 *sqlite*3_*close* 函数来释放相关资源. 除此之外, 在将商品信息通过 *sqlite*3_*exec* 函数向数据库文件插入之前, 需调用 *sqlite*3_*mprintf* 函数保存商品信息至固定格式的字符串中. 由于此字符串写在动态分配的内存块中, 因此当执行相关指令后, 应及时调用 *sqlite*3_*free* 函数释放内存. 然而在图 4 展示的 *insertItem* 函数中, 第 8 行、第 9 行 *sqlite*3_*close* 和 *sqlite*3_*free* 函数均未被调用. 实验过程中, 在不断添加新商品信息至数据库文件的 10 s 时间内, 该运行程序所占内存显著地从 5 MB 增加到 20 MB. 如果不采取干预措施, 程序的长时间运行将会占用大量机器内存, 导致其他程序无法正常运行.

```
 1 void insertItem(char* dbname){
 2   sqlite3* db; int rc; char* sql; int newItem=1;
 3   while(newItem){
 4     rc=sqlite3_open(dbname, &db);
 5     sql=sqlite3_mprintf("insert into item (name, price) values
 6                        ('book','1');");
 7     rc=sqlite3_exec(db, sql, 0, 0, 0);
 8     //sqlite3_free(sql);
 9     //sqlite3_close(db);
10     scanf("%d", &newItem);
11   }
12 }
```

图 4 程序 *invertory* 中的 *insertItem* 函数

针对上述 *insertItem* 函数中 API 调用序列不满足规约的情况, 我们使用以下两个 PPTL 公式表达期望的 API 调用序列规约, 表示在调用 API 函数 *sqlite*3_*open* 和 *sqlite*3_*mprintf* 后, 应分别调用 *sqlite*3_*close* 和 *sqlite*3_*free* 函数以释放相关资源:

$$\Box((sqlite3_open=1)\rightarrow\Diamond(sqlite3_close=1))$$

$$\Box((sqlite3_mprintf=1)\rightarrow\Diamond(sqlite3_free=1))$$

从本节两个例子能够看出以下 3 点.

(1) 程序运行中违反 API 调用序列规约, 有可能在大多数情况下并不会造成直接影响. 例如, 第 2.1 节程序 *fast_vacuum* 中的 *execSql* 函数在正常情况下可以运行完成并产生预期输出;

(2) 违反 API 调用序列规约所造成的后果有可能是不易察觉的. 例如在第 2.2 节程序 *invertory* 的 *insertItem* 函数中, *sqlite3_close* 和 *sqlite3_free* 函数的调用缺失会缓慢地累积内存泄漏;

(3) 应在程序运行过程中尽早发现违反 API 调用序列规约行为, 否则, 随时可能发生的电源故障异常会导致第 2.1 节程序 *fast_vacuum* 中数据库结果不一致, 以及随时可能出现的内存耗尽会导致第 2.2 节程序 *invertory* 和其他相关程序崩溃.

综上分析,有必要在程序运行过程中开展对 SQLite3 数据库 API 函数调用序列的高效运行时验证, 及时发现违反调用序列规约的情况.

## 3 API 调用序列规约的自动挖掘方法

为了用户能够更好地正确完成 API 函数调用, SQLite3 数据库提供了每个 API 函数的自然语言形式的详细描述(www.sqlite.org/c3ref/funclist.html), 包括参数与返回值类型以及具体实现功能等. 对于某项功能需要若干 API 函数合作才能完成的情况, 由于调用序列的正确性对功能实现的完整性至关重要, 正确的调用序列往往会以“应该”(should)、“必须”(must)等词语加以强调. 基于此特征, 我们开发了用以挖掘 SQLite3 官方网站所提供 API 调用序列规约的自动化工具, 其框架如图 5 所示.

API 调用序列规约的自动挖掘框架主要包括如下 5 个步骤.

(1) 将初始待分析网址输入至前端输入模块;

(2) 网站访问模块自动获取网站内容;

(3) 网站页面中包含“must”和“should”的相关语句以及包含的网址链接分别由内容提取模块提取和链接提取模块提取存储;

(4) 链接过滤器将目前正在分析的网址添加至已访问 URL 库中, 并对链接提取模块获取的网址进行过滤, 将尚未访问过的网址存储至未访问 URL 库中;

(5) 链接过滤器判断未访问 URL 库是否为空: 若为空, 则表明所有可访问网址均已被分析, 整个过程结束; 否则, 从库中取出首条网址, 并输入至网站访问模块, 开启新一轮调用序列规约挖掘.

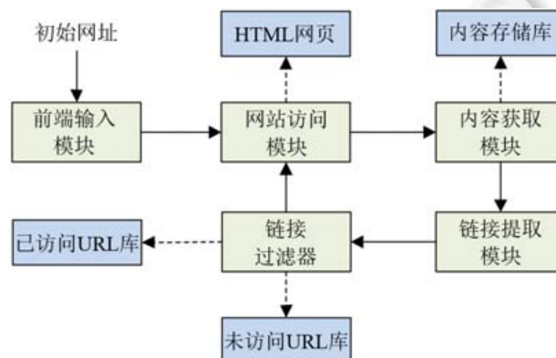


图 5　API 调用序列规约自动挖掘框架

以输入网址“www.sqlite.org/c3ref/open.html”为例, 图 6 展示了自动挖掘框架中, 每个模块处理得到的相关内容. 其中, 内容获取模块获得包含“must”和“should”关键字的 9 条 API 函数描述语句. 由于空间限制, 图中只显示前两条, 第 1 条与 API 调用序列规约相关, 不相关语句以下划线表示. 链接提取模块将当前网址中能够访问到的 5 条网址链接提取出来, 在经过链接过滤器分析后, 将两条尚未访问的链接存储至未访问 URL 库中(已访问过的链接在图中以下划线表示), 以便进行新一轮分析.

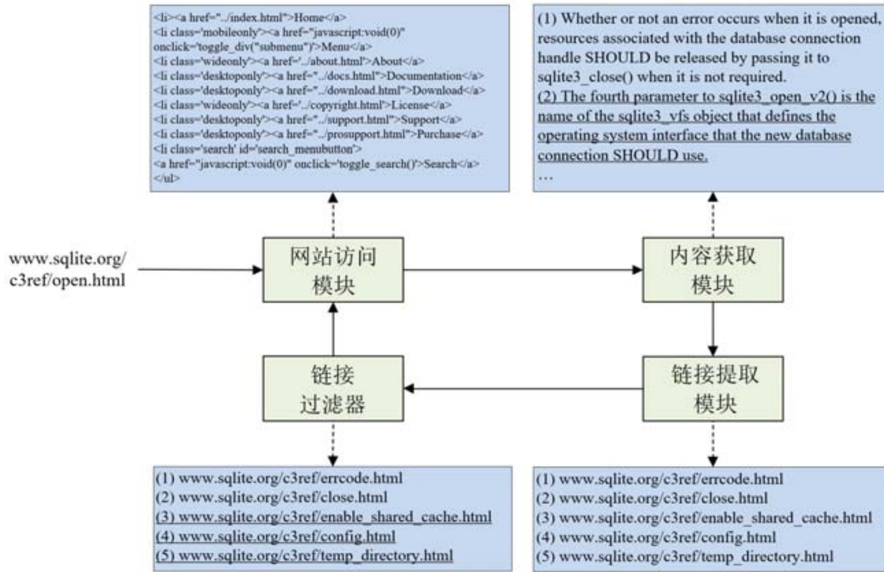


图 6　API 调用序列规约自动挖掘结果示例

利用上述所提出的 API 调用序列规约自动挖掘框架, 我们从 SQLite3 官方网站的 API 函数描述文档中共提取出 197 条含有“must”和“should”的语句. 但如图 6 例子所示, 并非所有提取出的语句均与 API 调用序列规约相关, 因此需要进一步人工核查以准确筛选出 API 调用序列规约. 最终, 我们选择出 12 条 API 调用序列规约并将其形式化表达为 PPTL 公式, 构建出如图 7 所示规约库. 其中, 前 3 条为 LTL 和 CTL 公式不能表达的周期重复规约, 分别与 *step*, *backup* 和 *initialize* 操作相关, 其余 9 条为安全性或弱公平性规约.

| API调用规约 | PPTL公式 |
| --- | --- |
| (1) After a prepared statement has been prepared, *sqlite3_step* **must** be called to evaluate the statement. The application **must** finalize every prepared statement to avoid resource leaks. | $\Diamond(((sqlite3_prepare_v2=1);(sqlite3_step=1);(sqlite3_finalize=1))^{+})$ |
| (2) There **should** be exactly one call to *sqlite3_backup_finish* for each successful call to *sqlite3_backup_init*. *sqlite3_backup_step* is called one or more times to transfer the data between the two databases. | $\Diamond(((sqlite3_backup_init=1);(sqlite3_backup_step=1);(sqlite3_backup_finish=1))^{+})$ |
| (3) The *sqlite3_shutdown* routine **should** be called to deallocate any resources that were allocated by *sqlite3_initialize*. | $\Diamond(((sqlite3_initialize=1);(sqlite3_shutdown=1))^{+})$ |
| (4) Whether or not an error occurs when it is opened, resources associated with the database connection handle **should** be released by passing it to *sqlite3_close* when it is not required. | $\Box((sqlite3_open=1)\rightarrow\Diamond(sqlite3_close=1))$ |
| (5) The strings returned by *sqlite3_mprintf* and *sqlite3_vmprintf* routines **should** be released by *sqlite3_free*. | $\Box(((sqlite3_vmprintf=1)\vee(sqlite3_mprintf=1))\rightarrow\Diamond(sqlite3_free=1))$ |
| (6) You **should** call function *sqlite3_column_text* or *sqlite3_column_blob* first to force the result into the desired format, then invoke function *sqlite3_column_bytes* to find the size of the result. | $\Box((sqlite3_column_bytes=1)\rightarrow((sqlite3_column_text=1)\vee(sqlite3_column_blob=1)))$ |
| (7) After the application has finished with the result from *sqlite3_get_table*, it **must** pass the result table pointer to *sqlite3_free_table* in order to release the memory that was malloced. | $\Box((sqlite3_get_table=1)\rightarrow\Diamond(sqlite3_free_table=1))$ |
| (8) The string returned by *sqlite3_expanded_sql* **must** be freed by the application by passing it to function *sqlite3_free*. | $\Box((sqlite3_expanded_sql=1)\rightarrow\Diamond(sqlite3_free=1))$ |
| (9) Extension loading **must** be enabled using *sqlite3_enable_load_extension* prior to calling function *sqlite3_load_extension*, otherwise an error will be returned. | $\Box((sqlite3_load_extension=1)\rightarrow(sqlite3_enable_load_extension=1))$ |
| (10) The *sqlite3_snapshot* object returned from a successful call to *sqlite3_snapshot_get* **must** be freed using *sqlite3_snapshot_free* to avoid a memory leak. | $\Box((sqlite3_snapshot_get=1)\rightarrow\Diamond(sqlite3_snapshot_free=1))$ |
| (11) To avoid a resource leak, every open BLOB handle **should** eventually be released by a call to function *sqlite3_blob_close*. | $\Box((sqlite3_blob_open=1)\rightarrow\Diamond(sqlite3_blob_close=1))$ |
| (12) *sqlite3_free* **should** be called with a pointer previously returned by *sqlite3_malloc* or *sqlite3_realloc* to release that memory so that it might be reused. | $\Box(((sqlite3_malloc=1)\vee(sqlite3_realloc=1))\rightarrow\Diamond(sqlite3_free=1))$ |

图 7　API 调用序列规约和相应的 PPTL 公式

## 4 SQLite3 数据库 API 调用序列的运行时验证

针对 C 程序运行过程中执行的 SQLite3 数据库 API 调用序列，本文基于文献[25,26]中所提出的并行运行时验证框架进行运行时监控，验证框架相同部分在此不再赘述，下面将介绍程序插桩模块和验证模块中对已有工作的改进.

### 4.1 程序插桩模块

在前端模块对 C 程序进行语法分析后，我们对程序语法树进行插桩. 与已有方法不同的是，在程序执行过程中，主要关注图 7 中 PPTL 公式涉及的 API 函数. 由于每次 API 函数被调用时，其可能对不同对象进行操作，为了区分这些调用，每个 API 函数需绑定一个特定参数作为其目标对象，当其被调用时,目标对象的地址也被记录下来. 只有当 API 函数和目标对象地址均一致时，才被认为是相同调用. 表 1 显示了图 7 中每个 API 函数对应的目标对象及其简要描述.

表 1 图 7 中每个 API 函数对应的目标对象及其简要描述

| API 函数名称 | 目标对象 | 简要描述 |
|---|---|---|
| *sqlite3_open* | *sqlite3 **db* | 数据库连接句柄 |
| *sqlite3_close* | *sqlite3 *db* | 数据库对象 |
| *sqlite3_mprintf* | *const char *p* | 分配内存的初始地址 |
| *sqlite3_vmprintf* | *const char *p* | 分配内存的初始地址 |
| *sqlite3_free* | *void *p* | 分配内存的初始地址 |
| *sqlite3_column_bytes* | *sqlite3_stmt *pStmt* | SQL语句 |
| *sqlite3_column_text* | *sqlite3_stmt *pStmt* | SQL语句 |
| *sqlite3_column_blob* | *sqlite3_stmt *pStmt* | SQL语句 |
| *sqlite3_prepare_v2* | *sqlite3_stmt **ppStmt* | SQL语句句柄 |
| *sqlite3_step* | *sqlite3_stmt *pStmt* | 准备好的SQL语句 |
| *sqlite3_finalize* | *sqlite3_stmt *pStmt* | 准备好的SQL语句 |
| *sqlite3_get_table* | *char ***pazResult* | 查询结果 |
| *sqlite3_free_table* | *char **result* | 指向结果表的指针 |
| *sqlite3_expanded_sql* | *sqlite3_stmt *pStmt* | SQL语句 |
| *sqlite3_load_extension* | *sqlite3 *db* | 数据库对象 |
| *sqlite3_enable_load_extension* | *sqlite3 *db* | 数据库对象 |
| *sqlite3_snapshot_get* | *sqlite3_snapshot *p* | snapshot对象 |
| *sqlite3_snapshot_free* | *sqlite3_snapshot *p* | snapshot对象 |
| *sqlite3_backup_init* | *sqlite3_backup *p* | backup对象 |
| *sqlite3_backup_finish* | *sqlite3_backup *p* | backup对象 |
| *sqlite3_blob_open* | *sqlite3_blob *pBlob* | BLOB句柄 |
| *sqlite3_blob_close* | *sqlite3_blob *pBlob* | BLOB句柄 |
| *sqlite3_malloc* | *void *p* | 分配内存的初始地址 |
| *sqlite3_realloc* | *void *p* | 分配内存的初始地址 |
| *sqlite3_initialize* | *void* | 无参数 |
| *sqlite3_shutdown* | *void* | 无参数 |

当相关 API 函数被调用或返回时，插桩函数 *StoreAP* 也会同时被调用，以存储该状态下相应原子命题的真值. 为了更加清晰，API 函数名与其对应的原子命题具有相同名称. 例如，*StoreAP*(“*sqlite3_open*”,1,*st*,“0x01”)被插入到函数 *sqlite3_open* 被调用之后的位置，而 *StoreAP*(“*sqlite3_open*”,0,*st*,“0x01”)被插入到函数 *sqlite3_open* 返回之后的位置，其中，*st* 代表当前程序状态的索引，“0x01”代表函数 *sqlite3_open* 所对应的目标对象地址. 函数 *StoreAP* 的细节在算法 1 中给出.

**算法 1**. *StoreAP*(*p*,*v*,*st*,*ad*).

输入：原子命题 *p*、原子命题 *p* 的真值 *v*、程序状态索引 *st*、目标对象地址 *ad*;

输出：存储原子命题初始真值的容器 *iv*、存储原子命题真值变化时状态索引的容器 *tr*、存储原子命题最新真值的容器 *lv*.

01: **IF** *p* 是首次处理 **THEN**

```
02:     iv[ad,p]=v;                       /*存储原子命题 p 的初始真值*/
03:     tr[ad,p].push_back(st);       /*存储原子命题 p 真值变化时的状态索引*/
04:     lv[ad,p]=v;                       /*存储原子命题 p 的最新真值*/
05:  ELSE IF v!=lv[ad,p] THEN       /*只有当原子命题 p 的真值发生改变,才有以下操作*/
06:     tr[ad,p].push_back(st);       /*存储原子命题 p 真值变化时的状态索引*/
07:     lv[ad,p]=v;                       /*存储原子命题 p 的最新真值*/
08:  END
```

### 4.2 运行时验证模块

为了更好地利用多核机器的硬件计算资源以提高效率, 本文利用并行运行时验证方法, 同时运行待验证程序的执行模块和针对 API 函数调用序列的验证模块, 对程序动态执行过程中 API 函数的调用序列进行实时监控, 验证模块运行流程如图 8 所示.

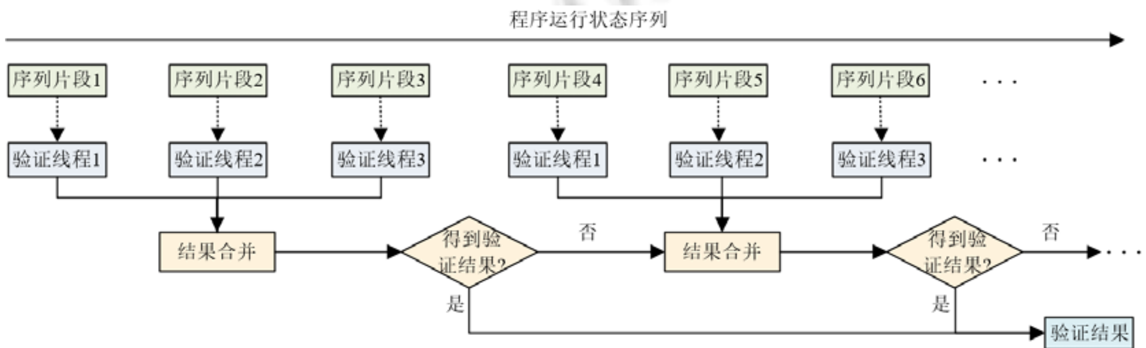


图 8　验证模块运行流程

图 9 展示了验证模块中不同子模块之间的调用关系, 包括 *RunTimeCheck*, *Segment*, *LNFGPath* 和 *ResultMerge* 这 4 个子模块. 作为最上层子模块,*RunTimeCheck* 负责对整个状态序列的验证, 它将多个序列片段的验证任务交给同时执行的子模块 *Segment*. *Segment* 为指定序列片段分配验证线程, 并调用子模块 *LNFGPath* 来探索 LNFG 中对应于该序列片段的可扩展路径. 子模块 *ResultMerge* 将 *Segment* 得到的不同序列片段的验证结果进行合并, 获得最终验证结果后, 将其返回给最上层的 *RunTimeCheck* 子模块.

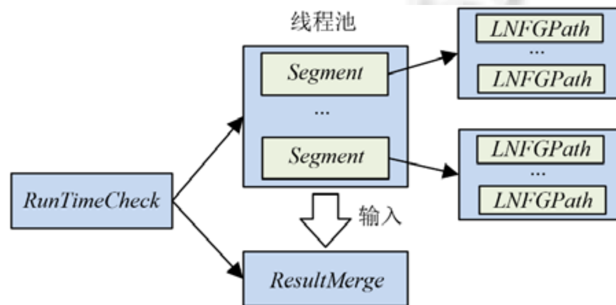


图 9　不同子模块之间的调用关系

## 5　工具实现与实验评估

基于前文提出的调用序列规约挖掘和并行运行时验证方法, 我们开发了自动化验证工具 SQLite3Check, 其以调用 SQLite3 数据库 API 函数的 C 程序为输入, 在程序运行时, 自动监控程序执行轨迹是否满足图 7 中 PPTL 公式表达的 API 调用序列规约.

我们使用装有 64 位系统的机器对 SQLite3Check 展开实验, CPU 型号为四核的 Intel i5, 主频为 2.3 GHz, 机器内存为 8 GB. 在验证过程中, 同时运行 4 个验证线程, 每个序列片段包含 $10^5$ 个状态. 该实验为了回答以下两个问题:

(1) 基于本文建立的 API 调用序列规约库, SQLite3Check 工具能否准确发现 API 调用序列违反相应规约?

(2) 本文提出的并行运行时验证方法在多核系统中能否有效提高验证效率?

首先, 为了评估方法及工具的有效性, SQLite3Check 对表 2 中描述的 30 个调用 SQLite3 数据库 API 函数的 C 程序进行运行时验证, 其中, 前 12 个程序为开源网站 GitHub 上发布的程序(https://github.com), 其余 18 个来自 SQLite 版本 3.36.0 的源代码文件夹(https://sqlite.org/download.html), 用于测试 SQLite3 源代码. 表 2 中, “验证时间”列给出了 SQLite3Check 为每个 API 函数调用序列规约消耗的平均验证时间(ms), “验证结果”列说明程序是否满足所有规约, “√”表示所有规约均被满足, “×”表示至少违反一个规约. 如果某个或多个规约被违反, 则在括号中列出该规约在图 7 中对应的序号.

表 2 SQLite3Check 的验证结果

| 程序名称 | 代码行数 | 简要描述 | 程序状态数 | 验证时间 | 验证结果 |
|---|---|---|---|---|---|
| addressBook | 582 | 通讯簿 | $8.2\times10^6$ | 516 | × (1) |
| ATM | 560 | ATM服务器与客户端 | $3.5\times10^7$ | 1 755 | √ |
| basicCApi | 301 | 基本 API 函数调用 | $6.3\times10^5$ | 117 | √ |
| college | 383 | 学校教务管理 | $7.5\times10^7$ | 1 920 | √ |
| invertory | 598 | 商品库存管理 | $4.5\times10^6$ | 280 | × (4, 5) |
| multiThreadStress | 222 | 多线程读写 | $3.5\times10^6$ | 187 | × (4) |
| passwordLock | 420 | 加密解密 | $6.0\times10^6$ | 256 | × (7) |
| speedTest | 426 | 读写速度测试 | $3.4\times10^6$ | 304 | √ |
| staffManagement | 607 | 员工登记系统 | $6.7\times10^6$ | 618 | × (7) |
| StudentManagement | 248 | 分数管理系统 | $5.7\times10^6$ | 258 | × (7) |
| taskReminder | 502 | 任务提醒 | $1.3\times10^7$ | 481 | × (4) |
| trooperData | 221 | 攻击防御模拟 | $1.0\times10^7$ | 328 | × (1) |
| dbfuzz | 514 | 数据库模糊测试 | $1.0\times10^7$ | 327 | √ |
| dbhash | 376 | 文件SHA1加密 | $7.0\times10^6$ | 304 | × (6) |
| fast_vacuum | 161 | VACCUM优化 | $7.4\times10^5$ | 132 | × (1) |
| fuzzcheck | 831 | 数据库回归测试 | $7.0\times10^6$ | 561 | × (6) |
| fuzzershell | 800 | 模糊测试 | $8.4\times10^6$ | 280 | × (6) |
| kvtest | 1 108 | 键值对性能测试 | $1.9\times10^7$ | 609 | × (5, 6) |
| loadfts | 245 | FTS性能测试 | $5.5\times10^6$ | 234 | √ |
| max-limits | 47 | 最大值展示 | $9.0\times10^6$ | 397 | √ |
| offsets | 694 | 表内文件查找 | $2.0\times10^7$ | 748 | × (5) |
| rollback-test | 149 | 数据库回滚 | $1.3\times10^7$ | 514 | × (4) |
| showdb | 2 096 | 文件输出 | $2.8\times10^7$ | 1 309 | √ |
| showstat4 | 470 | sqlite_stat4表展示 | $3.0\times10^7$ | 771 | √ |
| speedtest1 | 1471 | 性能测试 | $4.7\times10^7$ | 1 497 | √ |
| speedtest8inst1 | 239 | 性能测试 | $9.6\times10^6$ | 304 | √ |
| speedtest8 | 220 | 性能测试 | $1.3\times10^7$ | 421 | √ |
| speedtest16 | 182 | UTF16性能测试 | $1.4\times10^6$ | 93 | √ |
| sqldiff | 1 140 | 数据库文件比较 | $7.9\times10^7$ | 280 | ×(5) |
| wordcount | 919 | 单词插入 | $7.1\times10^7$ | 256 | √ |
| 总计 | 16 732 | 30个 | $5.5\times10^8$ | 16 057 | 30 (100%) |

从表 2 可以看出, 大多数程序可以在不到 1 秒的时间内得到验证. 在 30 个验证程序中, 违背 API 函数调用序列规约的数量达到一半, 包括 GitHub 上 12 个程序中的 8 个以及 SQLite 源代码文件夹中 18 个程序中的 8 个. 在由不熟悉 API 函数调用规约的程序员编写的 GitHub 程序中, 违反率高达 66.7%. 更糟糕的是: 尽管 SQLite 源代码文件夹中的程序均由开发 SQLite 数据库本身的程序员编写, 但 API 函数调用序列规约的违反率仍高达 44.4%. 表 2 所示的验证结果表明, 对调用 SQLite3 数据库 API 函数的程序进行调用序列验证是十分必要的.

进一步, 为了证明在多核系统中本文提出的并行验证方法对验证性能的提升, 我们还实现了一个基于传

www.jos.org.cn

统串行方法的验证工具 SQLite3Check*, 并针对图 7 的 12 条 API 函数调用序列规约和表 2 的 30 个 C 程序开展运行时验证, 将其与 SQLite3Check 工具进行效率比较.

图 10 展示了分别使用 SQLite3Check*和 SQLite3Check 工具验证每个规约所消耗的平均时间的对比. 其中, $x$ 轴显示每个程序的名称, $y$ 轴显示完成验证任务所消耗的时间(ms), 两种柱形分别为 SQLite3Check*和 SQLite3Check 所消耗时间. 从实验结果可以看出: 在多核系统中, 采用本文提出的并行验证方法能够有效提高针对 C 程序的 SQLite3 数据库 API 调用序列的运行时验证效率.

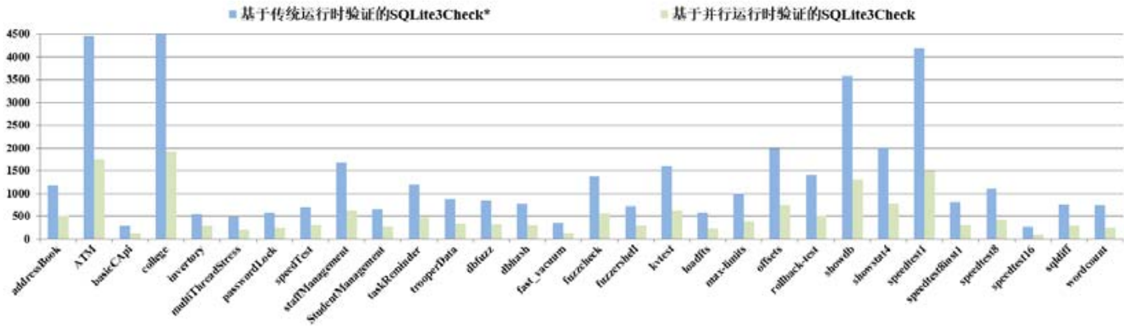


图 10 并行运行时验证方法的效率提升

在对结果进一步分析后还能发现: 尽管 4 个验证线程被用来并行地验证不同序列片段, 但 SQLite3Check 的验证时间仅减少为 SQLite3Check*的约 1/3 而非 1/4. 这是由于从处理器底层实现来看, 由于处理器核数限制, 4 个验证线程并非完全独立并行运行. 此外, 除了每个验证线程针对相应序列片段所花费的验证时间外, 总的验证时间还包括只能以串行方式在调度线程中完成的任务分配与结果合并所花费的时间.

## 6 相关工作

与本文相关的研究工作主要包括 API 函数调用序列验证和数据库验证这两个方面.

### 6.1 API函数调用序列验证

API 函数调用序列验证已经应用于 Java 程序、设备驱动程序和 Web 服务等领域, 其中, 静态分析和运行时验证是两种最常用的方法. 在 Li 等人[5]提出的用于验证 Java 程序 API 函数调用序列的框架中, 原始程序被插桩以支持探索包括异常控制流在内的所有可能控制流. Jass(http://modernjass.sourceforge.net/)是一个预编译工具, 通过使用能够在运行时检查的规约对 Java 程序进行注释, 监视顺序对象的动态行为和函数调用的顺序. 基于类型状态检查方法[27], DeLine 等人开发了静态软件分析工具 Fugue[7], 用于检测 Java 程序是否符合 API 调用序列规约. Post 等人[9]从 Linux 文档中提取 API 函数调用规约, 并用扩展语言 SLICx 形式化这些规约, 以 CBMC 限界模型检查器为验证后端, 检测死锁和内存泄漏. 对于 Linux 标准库, Olver 项目(http://linuxtesting.org)中开发了一个规约检查工具, 验证 Linux 内核和用户级软件间的调用规约满足性. 针对 Web 服务, Hallé 等人开发了协议控制器[11], 以检查 Web 服务中的实际服务器实现是否满足其文档中描述的函数调用规约. Ramsokul 和 Sowmya[12,13]将 Web 服务协议模型转化为 Promela 语言, 提出了一种验证 Web 服务协议实现是否符合协议规约的方法, 能够利用 SPIN 工具验证协议的正确性.

### 6.2 数据库验证

目前已有大量研究针对数据库系统的事务安全性开展, 将程序验证应用于事务的研究可以追溯到倡导使用霍尔逻辑或最弱前提的开创性工作[28,29]. 考虑到数据库事务的完整性维护问题, Benedikt 等人[30]研究了最弱前提的理论性质, 使用一阶逻辑及其扩展形式来指定事务和查询功能. 针对对象数据库中的动态规约, Benzaken 等人[31]将其形式化描述为一阶 LTL 公式, 并结合定理证明与静态分析方法, 验证给定事务的执行是

否符合动态完整性规约. 为了保证 ACID 性质, Alagic 和 Fazeli[32]结合静态和动态验证方法, 构建基于面向对象的数据库管理系统实现的 ACID 事务. 此外, 还有一些研究是为了构建完整的高可靠性数据库系统. Benzaken 等人[33]在证明辅助工具 Coq 的帮助下, 提出一个 SQL 代数以形式化定义主流物理运算符, 证明实际 SQL 引擎物理层. 通过定义系统行为功能规约, Malecha 等人[34]构建了一个经过充分验证的轻量级关系数据库管理系统, 并在 Coq 工具中验证了系统实现.

综上可知, 虽然在 API 函数调用序列验证和数据库验证两个方面均已有较为深入的研究, 但在数据库系统的 API 函数调用序列验证方面尚未有成熟研究. 此外, 现有的 API 函数调用序列验证大都仅注重发现的规约违反数量, 对验证效率关注不足. 为了弥补该领域的研究空白, 本文提出了针对 SQLite3 数据库 API 函数调用序列规约的自动挖掘框架以及并行运行时验证方法, 高效地动态监控程序在运行过程中对 API 函数的调用是否满足指定规约.

## 7 总结与展望

针对 SQLite3 数据库的多个 API 函数在调用过程中, 其调用序列是否满足指定规约的问题, 本文首先针对 SQLite3 数据库 API 描述文档提出调用序列规约自动挖掘框架, 提取相关信息以辅助人工将其形式化表达为 PPTL 公式, 进而使用运行时验证方法实时监控调用 API 函数的 C 程序的动态运行状态. 在验证过程中, 为了加快验证速度, 提出了多任务调度策略以将程序运行时产生的状态序列划分为若干序列片段, 并利用创建的多个线程对这些序列片段进行并行验证, 最后将各个片段的验证结果进行合并, 得到最终的验证结果. 实验结果表明: 本文所提方法能够发现 30 个被验证程序中的 16 个违反 API 函数调用序列规约, 包括 SQLite3 源代码包含的 8 个测试程序和 GitHub 网站上的 8 个公开程序. 此外的对比实验表明: 本文提出的并行运行时验证方法能够更为有效发挥多核系统的硬件能力, 提高验证效率.

然而, 由于单个 CPU 的计算能力存在瓶颈, 验证线程的不断创建会加重线程调度负载, 从而降低验证效率. 为了能够同时高效地验证更多的序列片段, 在今后的工作中, 我们将研究如何将分布式网络应用到程序的并行运行时验证中, 以进一步提高大规模程序的验证效率.

**References**:


[1] Lee K, Won Y. Smart layers and dumb result: IO characterization of an Android-based smartphone. In: Proc. of the 10th ACM Int'l Conf. on Embedded Software. Tampere: ACM, 2012. 23–32.

[2] Haldar S. Inside Sqlite. O'Reilly Media Inc., 2007.

[3] Nadi S, Krüger S, Mezini M, *et al*. Jumping through hoops: Why do Java developers struggle with cryptography APIs? In: Proc. of the 38th Int'l Conf. on Software Engineering. Austin: ACM, 2016. 935–946.

[4] Li Z, Wu JZ, Li MS. Study on key issues in API usage. Ruan Jian Xue Bao/Journal of Software, 2018, 29(6): 1716–1738 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5541.htm [doi: 10.13328/j.cnki.jos.005541]

[5] Li X, Hoover HJ, Rudnicki P. API conformance verification for Java programs. In: Proc. of the 12th Int'l Conf. on Formal Engineering Methods. Shanghai: Springer, 2010. 188–203.

[6] Rieken J. Design by contract for Java-revised [MS. Thesis]. Department fürInformatik, Universität Oldenburg, 2007.

[7] DeLine R, Fähndrich M. Typestates for objects. In: Proc. of the 18th European Conf. on Object-Oriented Programming. Oslo: Springer, 2004. 465–490.

[8] Bierhoff K, Beckman NE, Aldrich J. Practical API protocol checking with access permissions. In: Proc. of the 23rd European Conf. on Object-Oriented Programming. Genoa: Springer, 2009. 195–219.

[9] Post H, Küchlin W. Integrated static analysis for Linux device driver verification. In: Proc. of the 6th Int'l Conf. on Integrated Formal Methods. Oxford: Springer, 2007. 518–537.

[10] Rubanov V, Silakov D. Certification infrastructure for the Linux standard base (LSB). In: Proc. of the 2nd Int'l Workshop on Foundations and Techniques for Open Source Software Certification. 2008. 79.

[11] Hallé S, Bultan T, Hughes G, *et al*. Runtime verification of Web service interface contracts. Computer, 2010, 43(3): 59–66.

[12] Ramsokul P, Sowmya A. Aseha: A framework for modelling and verification ofweb services protocols. In: Proc. of the 4th IEEE Int’l Conf. on Software Engineering and Formal Methods. Pune: IEEE, 2006. 196−205.

[13] Ramsokul P, Sowmya A. A sniffer based approach to WS protocols conformance checking. In: Proc. of the 5th Int’l Symp. on Parallel and Distributed Computing. Timisoara: IEEE, 2006. 58−65.

[14] Bartocci E, Falcone Y, Francalanza A, *et al*. Introduction to runtime verification. In: Lectures on Runtime Verification. Springer, 2018. 1−33.

[15] Meredith PON, Jin D, Griffith D, *et al*. An overview of the MOP runtime verification framework. Int’l Journal on Software Tools for Technology Transfer, 2012, 14(3): 249−289.

[16] Li XS, Tao XP, Song W. Runtime verification of spatio-temporal properties for pervasive computing applications. Ruan Jian Xue Bao/Journal of Software, 2018, 29(6): 1622−1634 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5466.htm [doi: 10.13328/j.cnki.jos.005466]

[17] Basin D, Klaedtke F, Zălinescu E. Runtime verification of temporal properties over out-of-order data streams. In: Proc. of the 29th Int’l Conf. on Computer Aided Verification. Heidelberg: Springer, 2017. 356−376.

[18] Pnueli A. The temporal logic of programs. In: Proc. of the 18th Annual Symp. on Foundations of Computer Science. Rhode Island: IEEE, 1977. 46−57.

[19] Clarke EM, Emerson EA. Design and synthesis of synchronization skeletons using branching time temporal logic. In: Proc. of the Workshop on Logic of Programs. New York: Springer, 1981. 52−71.

[20] Leucker M, Schallhart C. A brief account of runtime verification. The Journal of Logic and Algebraic Programming, 2009, 78(5): 293−303.

[21] Falcone Y, Havelund K, Reger G. A tutorial on runtime verification. Engineering Dependable Software Systems, 2013, 34: 141−175.

[22] Tian C, Duan ZH. Propositional projection temporal logic, Buchi automata and $\omega$-regular expressions. In: Proc. of the 5th Int’l Conf. on Theory and Applications of Models of Computation. Xi’an: Springer, 2008. 47−58.

[23] Duan ZH. An extended interval temporal logic and a framing technique for temporal logic programming [Ph.D. Thesis]. Newcastle: University of Newcastle Upon Tyne, 1996.

[24] Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation. In: Proc. of the 2nd Int’l Symp. on Code Generation and Optimization. San Jose: IEEE, 2004. 75−88.

[25] Yu B. Research on efficient runtime verification for MSVL programs [Ph.D. Thesis]. Xi’an: Xidian University, 2019.

[26] Yu B, Zhang N, Lu X, *et al*. Runtime verification approach for DoS attack detection in edge servers. Journal on Communications, 2021, 42(9): 75−86 (in Chinese with English abstract). http://www.infocomm-journal.com/txxb/CN/10.11959/j.issn.1000-436x.2021169

[27] Strom RE, Yemini S. Typestate: A programming language concept for enhancing software reliability. IEEE Trans. on Software Engineering, 1986, 12(1): 157−171.

[28] Gardarin G, Melkanoff M. Proving consistency of database transactions. In: Proc. of the 5th Int’l Conf. on Very Large Data Bases. Rio de Janeiro: IEEE, 1979. 291−298.

[29] Casanova MR, Bernstein PA. A formal system for reasoning about programs accessing a relational database. ACM Trans. on Programming Languages and Systems, 1980, 2(3): 386−414.

[30] Benedikt M, Griffin T, Libkin L. Verifiable properties of database transactions. Information and Computation, 1998, 147(1): 57−88.

[31] Benzaken V, Cerrito S, Praud S. Static verification of dynamical integrity constraints: A semantics based approach. Networking and Information Systems Journal, 2000.

[32] Alagić S, Fazeli A. Verifiable object-oriented transactions. In: Proc. of the Concurrent Objects and Beyond. Springer, 2014. 251−275.

[33] Benzaken V, Contejean E, Keller C, *et al*. A Coq formalisation of SQL’s execution engines. In: Proc. of the 9th Int’l Conf. on Interactive Theorem Proving. Oxford: Springer, 2018. 88−107.

[34] Malecha G, Morrisett G, Shinnar A, *et al*. Toward a verified relational database management system. In: Proc. of the 37th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. Madrid: ACM, 2010. 237–248.

**附中文参考文献:**

[4] 李正, 吴敬征, 李明树. API 使用的关键问题研究. 软件学报, 2018, 29(6): 1716–1738. http://www.jos.org.cn/1000-9825/5541.htm [doi: 10.13328/j.cnki.jos.005541]

[16] 李晅松, 陶先平, 宋巍. 普适计算应用时空性质的运行时验证. 软件学报, 2018, 29(6): 1622–1634. http://www.jos.org.cn/1000-9825/5466.htm [doi: 10.13328/j.cnki.jos.005466]

[25] 于斌. MSVL 程序的高效运行时验证方法研究 [博士学位论文]. 西安: 西安电子科技大学, 2019.

[26] 于斌, 张南, 陆旭, 段振华, 田聪. 基于运行时验证的边缘服务器 DoS 攻击检测方法. 通信学报, 2021, 42(9): 75–86. http://www.infocomm-journal.com/txxb/CN/10.11959/j.issn.1000-436x.2021169

**于斌**(1990－), 男, 博士, 讲师, CCF 专业会员, 主要研究领域为模型检测, 运行时验证.

**段振华**(1948－), 男, 博士, 教授, 博士生导师, CCF 会士, 主要研究领域为时序逻辑, 形式化方法, 高可信嵌入式系统.

**陆旭**(1985－), 男, 博士, 副教授, CCF 专业会员, 主要研究领域为智能规划, 模型检测, 程序验证.

**张南**(1984－), 女, 博士, 副教授, 博士生导师, CCF 高级会员, 主要研究领域为形式化验证, 模型检测.

**田聪**(1981－), 女, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为可信软件的基础理论与方法, 人工智能系统安全.

www.jos.org.cn