基于 Verds 的 C 语言子集的模型检测方法①

张兰兰 [1,2]

[1](中国科学院软件研究所 计算机科学国家重点实验室, 北京 100190)
[2](中国科学院大学, 北京 100190)

摘 要: 针对现今软件使用逻辑错误的问题越来越多的出现, 提出了对最流行最普遍的编程语言——C 语言子集的模型检测方法的研究. 采用基于 Verds 工具的模型, 运用 C 语言子集转化成 Verds 模型的算法, 结合 Verds 工具和 MAGIC 工具实现模型检测. 引入反例引导的抽象精化方法使模型检测解决状态爆炸的问题.

关键词: 模型检测; 转化; Verds; CEGAR; MAGIC

Model Checking Method on Subset of C Language Based on Verds

ZHANG Lan-Lan[1,2]

[1](State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)
[2](Chinese Academy of Sciences, Beijing 100190, China)

Abstract: In problem of software logic errors, nowadays it emerges more and more. The paper presents the research for model checking methods of C language, that it is the most popular and general programming languages. The model checking based on Verds tools, using C language subset into Verds model algorithm, combined with the Verds tools and MAGIC tools. Introducing the counterexample guided abstraction refinement (CEGAR) method to solve the problem of state explosion.

Key words: model checking; tranform; Verds; CEGAR; MAGIC

随着人类对于软件的使用越来越来广泛普遍, 软件研究越来越深入, 对其精确度的要求也越来越高. 程序工作者对简单程序的逻辑可以很好做出正确的判断, 但是广泛应用在各个领域的程序每个都是庞大的架构, 繁多的结构分支, 这使得程序工作人员也无法梳理好这样程序的逻辑. 程序的语法错误可以在相应的平台上完成了检测, 但是并没有完成程序逻辑检测. 一个软件的是否完成项目的各项要求通过测试人员对其进行专业测试来验证, 达到各项测试要求的软件可以投入使用. 完成项目测试的软件, 我们不能说它是没有错误的, 更多的应用情况在软件日复一日的使用中出现, 那些被程序工作者忽视和没有被软件测试员测试出的问题将会凸显出来. 为了提高软件这方面的精确度, 科学家着手深入对程序准确性、稳定性和目的可达性的研究.

今天, C 语言是一个开放的语言, 语法形式多样, 不同的人可以编写出实现同一功能而源代码形式多样的程序. 使用 C 能编写高性能程序, 包括系统程序和应用程序. 但是 C 语言的逻辑验证问题还没有得到到解决, 这个问题在软件使用中有潜在的危险. 国家航天卫星的发射, 是不容失误的, 不容许出现问题的, 在卫星发射的过程中, 有很多涉及软件, 因为小小的逻辑问题的失误未被发现所引发的卫星发射的失败, 是我们已经经历过的事情. 为了减少这种的事情发生, 对软件, 对 C 语言逻辑问题的验证被大家提出. 基于 C 语言的模型检测是解决软件逻辑问题课题研究的一部分, 对于软件逻辑问题解决很有意义.

用模型的方法来描述和验证 C 程序可以发现软件应用中潜在的错误. 用模型描述和验证 C 程序是对其进行模型检测设计与实现的基础. 对 C 程序软件的逻辑验证的实现的正确性和完整性有重要的影响[1].

① 基金项目:国家科技重大专项(2012ZX01039-004)
收稿时间:2013-04-17;收到修改稿时间:2013-04-23

1 相关概念

程序模型的状态空间正比于程序的变量数目及变量的定义域, 即使很小的程序都可能具有巨大的状态空间, 甚至是无穷的, “状态空间爆炸”是在大规模软件工业设计中推广使用模型检验的一个主要障碍. 通过研究人员的努力, 许多方法被提出来, 如符号模型检验、规约技术、抽象技术等等. 其中抽象技术近年来开始得到人们关注, 逐步发展成为解决此种问题的一种重要手段.

1994 年, Clarke、Grumberg 和 Long 初步探索了应用抽象技术的模型检验方法, 通过构造程序的抽象模型来验证程序的性质. 然而, 该方法并没有完全解决“状态空间爆炸”问题. 抽象模型中的状态称为抽象状态. 相应地,抽象前的系统模型称为具体模型,具体模型中的状态称为具体状态. 要求抽象模型具有强保持性非常困难, 抽象通常会引入具体模型不存在的附加的行为. 当抽象模型不满足性质, 在程序的流程图中存在这样一条导致性质不满足的路径, 在程序中这样的一条路径是有语句组成的, 我们可以称语句组成的这条路径为“反例”, 我们可以肯定该反例存在于抽象模型, 但反例可能只是抽象模型中引入的一个附加行为, 不一定存在于具体模型, 这样的反例称为伪反例(spurious counterexample). 为解决这样的问题, Clarke 等人在之前的研究基础上继续深入, 提出了“反例引导的抽象精化(counterexample guided abstraction refinement, CEGAR)”[2]框架. 国外出现了很多对于C语言的模型检测和程序验证的研究, 如 Slam[3], Blast[4], Compcert[5], VARVEL[6]等. Slam 和 Blast 都采用了反例引导的抽象精化框架.

2 Verds验证工具

Verds[7-9]程序验证模型是由中科院软件所研究员张文辉老师提出并维护的. Verds 是验证层次化离散系统的模型验证工具. 系统的性质可用时序逻辑 CTL 来描述. 验证方法有两类, 其一是基于 CTL 限界语义的限界模型检测方法, 其二是基于三元布尔图的符号模型检测方法. 本论文选择 Verds 模型的原因, 这是因为 Verds 有足够的优势, 主要在以下两个方面:

(1) 实现在 VERDS 中的三元布尔图的模型检测方法和著名的 NuSMV2.5.0 符号模型检测工具进行过比较, 对两种类型的随机布尔程序的实验用例, VERDS 相比较于 NuSMV 有明显的优势; 对于一些协议模型, VERDS 与 NuSMV 及其他相关工具比较互有利弊. 另外, 限界模型检测和符号模型检测有一定的互补关系, 主要优势在于有些性质在一些系统中能够在较小的局部范围内得到验证或者反证. 使用VERDS进行验证有可能结合两者的长处.

(2) VERDS 使用的建模语言具有层次化的结构, 便于描述过程调用等, 与一般程序语言有一些共同之处, 适用于程序的描述. 并且验证方法能够应用“假设-保证”框架, 将较大的验证问题分解为较小的验证问题, 便于较大程序的验证.

Verds 模型一般有主模块和 Procedure 模块, 第二个模块和 C 语言中函数块类似. Verds 模型可以进行并行程序的验证, 这种情况会用到 Module 模块, 这个模块可以抽象一个线程. 性质的描述找SPEC模块, 用抽象树逻辑 CTL 表述.

CTL 是一种分支时态逻辑. CTL 公式的时态操作符由路径算子和时态算子组成. 路径算子有 2 种: A 表示“适用于所有的路径”, E 表示“存在至少 1 条以上的路径”. 时态算子由 X(neXt time)算子、F(Future)算子、G(Global)算子、U(Until)算子组成.

3 C到Verds模型转化步骤和预期结果

对于所需要验证的 C 程序, 我们假设在语法上正确的, 本文提出针对 C 语言的模型检测方法, 首先在Verds工具中对C程序的验证, 减少了建立模型的工作; 其次对于简单的 C 语言程序, 给出将 C 语言子集转化到 Verds 模型的算法, 这样实现了对 C 语言程序在verds 工具上的自动验证; 然后针对大程序存在的状态空间爆炸问题, 提出结合CEGAR减少状态的方法; 最后, 对 CEGAR[10]实现过程中, 重要的反例生成部分, 运用 MAGIC[11]工具得到符号迁移系统 LTS 判断是否为伪反例, 对 CEGAR 的完成具有很好的辅助作用.

本文将可支持的 C 语言程序转化成 Verds 模型, 有以下预期结果:

① 将合理的 C 语言语法结构转化到 Verds 模型, 例如 C 语言子集中的函数调用、函数声明、结构体等.

② 对于 C 语言的‘不可译’部分, 给出合理的翻译方法, 例如循环语句, switch 语句, 指针等.

③ 给出干净的和明确的输出格式.

④ 经过转化的 C 语言的 Verds 模型保证可以在 Verds 工具中进行模拟和验证.

4 可支持的C程序例子

为了以更有效的友好的方式解释可以支持的 C 语言子集, 首先总结下 Verds 模型可以涵盖的 C 语言子集:

① 一个程序可能会用到#include, 来引用标准库, 这些标准库的编译是内部进行, 我们看不到, 所以不考虑标准库的引用问题. 如果想完成这些功能的模拟, 我们也可以用额外的时间手动模拟, 这些功能不在本文的限制范围内.

② C 程序不允许输入操作, 如 scanf, 因为 Verds 工具没有读取输入的功能. 带有输入功能的标准库里的函数也是不支持的.

③ 支持的数据结构是简单的整数、字符、数组、结构体(内部成员没有指针的)和指向整数型的指针. 以上类型都可以作为函数的参数类型. 指针的计算不支持.

④ 表达式可以经过一般的操作组合起来, 如+, -, %, <.

⑤ 赋值、函数调用、跳转、条件语句和循环语句都作为语句来处理.

⑥ 函数的定义、声明、调用和返回值都是被允许的.

表 1 给出可支持的 C 语言关键字.

表 1 “可译”C 关键字

int	支持
char	支持
double,signed, unsigned,long, short	支持，不过被转换成 int
void	支持
struct	支持特定的书写格式
const, static	支持
define	支持
switch-case-defaul, if-else	支持
for ,do ,while	支持
return	支持返回表达式是指针的变量和整数
goto	支持
break	支持
continue	支持
labeled	支持
labeled statement	支持

本文以蓝牙驱动[13]程序作为例子, 在 Verds 上完成蓝牙驱动并发程序的性质验证.

5 C子集与Verds模型转化语义

5.1 结构体和语句

在 C 语言中结构体是其很重要的部分, 他使 C 语言程序处理更加方便如表所示, 在 Verds 中一般把数据结构认为是 int 型, Verds 也可以处理字符 char. 在 Verds 模型中, 它接受的数据结构并不全面, 这也是很多模型验证工具难解决的部分, 数据结构的内部成员不方便有指针, 一般为 int 或 char. 图 1 中 C 程序的目的是为结构体中的 max 和 min 赋值较大的值和较小的值. 在 Verds 中用 record 来描述 struct, 它们的成员函数不变, 不过在 Verds 中用结构体数组来描述结构体对象, 并且它的数组大小是提前说明的, 图 1 中大小为 3. 对于结构体成员的引用, Verds 和 C 是一样的. Verds 模型中语句是用一个符号做特殊标记, 并对符号赋值来推用程序的运行, 这里和 C 语言语句有很大区别. Verds 中没有条件语句和循环语句, 我们可以通过 Verds 模型推动程序运行的特殊符号的赋值完成条件判断和语句循环, 如图 1 中 Verds 所示.

```
C:
#define n 3
struct person{
      int max;
      int min;
}a[n];
mian(){
    int x;
    int y=2;
    for(x=0;x<n;x++){
    if(y>x){
            a[x].min=x;
            a[x].max=y;
       }else{
            a[x].min=y;
            a[x].max=x;
    } } }

Verds:
VVM struct
DEFINE n 3
VAR
   x: int;
   y: int;
   a[n]:   record{min: int;  max int;}
   pc:   {s0,s1,s2,s3,s4,s5,s6,s7,s8,s9,s10};
INIT
  x=0;
  y=0;
  (for x in [1..n]): (a[x].min=0&a[x].max=0);
  pc=s0;
```

```
TRANS
   pc=s0:    (pc,y):=(s1,2);
   pc=s1:    (pc,x):=(s2,0);
   pc=s2&(x<n):       (pc):=(s3);
   pc=s3&(y>x):       (pc):=(s4);
   pc=s4:    (pc,a[x].min):=(s5,x);
   pc=s5:    (pc,a[x].max):=(s6,y);
   pc=s7&!(y>x):      (pc):=(s8);
   pc=s8:    (pc,a[x].min):=(s9,y);
   pc=s9:    (pc,a[x].max):=(s10,x);
   pc=s6:    (pc):=(s10);
   pc=s10:   (pc):=(s2);
   pc=s2&!(x<n):      RETURN;
SPEC
  AG((for x in [1..n]): ( a[x].min<a[x].max))
```

图 1　结构体和语句

5.2 指针

指针是进行 C 语言的重要部分，很多系统程都会涉及到指针的应用，指针对一些软件是必不可少的，指针是 C 语言发展的一个分界点. 但是在 Verds 模型中没有类似指针的结构. 因此，我们需要找到合适的方法对指针的定义、赋值、引用和其他的操作进行抽象. 下面我们给出有关 C 语言指针在 Verds 模型下的转化. 这是本文创新点之一.

首先指针一般变量包含更多的信息量，而且它的使用很方便，如图2所示，这是因为C预言中的指针包含一个地址和一个数据，这个地址有一个指向，怎样找到指针的地址或其指向是整个指针问题的重点. 在 Verds 中我们建立一个数组来模拟图 2 中指针的存储，

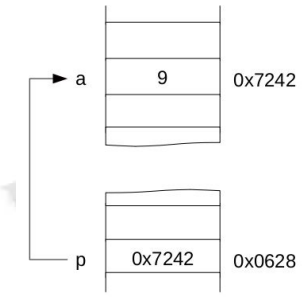

图 2　C 语言中指针

这样可以解决指向和数值的问题，我们将数组的下标作为指针的临时地址，数组的内容是指针的数值，为了扩展指针的操作，我们可以把数据元素定义为一个数据结构，图 3 显示了为解决指针的引用和指针赋值，采用结构体，结构体中的 mark 表示指针是否指向别的指针，而 addr 表示指向指针的地址. 在指针赋值时，检测 mark 值，如果为 0 则说明指针有引用，同时改变引用的数据值；否则只是更新本身的数据值. 本问不考虑很多指针指向同一地址的情况. 不同数据类型具有不同的结构体数组，因此不同的指针类型在模拟过程中有独立的数组定义.

```
    C :
    struct person{
        int age;
    };
    int main(){
        struct person p, *q, *r;
        q=&p;
        r=malloc(sizeof(struct person));
        p.age=24;
        q->age++;
        r->age=18;
        free(q);
        return   0;
    }
    Verds:
    VVM point
    VAR
        struct_p[1..26]:    record {age: int; addr: int; mark: 0..1;};

        pc:      {s0,s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,s11,s12
                 s13,s14,s15,s16,s17,s18};
    INIT
        (for x in [1..26]): struct_p[ x].addr=0&struct_p[x].mark=1;;
        pc=s0;
TRANS
      pc=s0:      (pc,struct_p[17 ].addr,struct_p[17].mark):=(s1,16,0);
      pc=s1:      (pc,struct_p[18 ].addr):=(s2,1);
      pc=s2:      (pc,struct_p[16 ].age):=(s3,24);
      pc=s3&!(struct_p[16].mark==1):      (pc):=(s4);
      pc=s4:      (pc,temp):=(s5 ,struct_p[16].addr);
      pc=s5:      (pc,struct_p[t emp].age):=(s6,struct_p[16].age);
      pc=s3& (struct_p[16].mark==1):      (pc):=(s6 );
      pc=s6:      (pc,struct_p[ 17].age):=(s7,struct_p[17].age+1);
      pc=s7&!(struct_p[17].mark==1):    (pc):=(s8);
      pc=s8:      (pc,temp):=(s9,struct_p[17].addr);
      pc=s9:      (pc,struct_p[temp].age):=(s10,struct_p[17].age);
      pc=s7&(struct_p[17].mark==1):     (pc):=(s10);
      pc=s10:     (pc,struct_p[18].age):=(s11,18);
      pc=s11&!(struct_p[18].mark==1):   (pc):=(s12);
      pc=s12:     (pc,temp):=(s13,struct_p[18].addr);
      pc=s13:     (pc,struct_p[temp].age):=(s14,struct_p[18].age);
      pc=s11&(struct_p[18].mark==1):    (pc):=(s14);
      pc=s14:     (pc,struct_p[17].age):=(s15,0);
      pc=s15&!(struct_p[17].mark):      (pc):=(s16);
      pc=s16:     (pc,temp):=(s17,struct_p[17].value);
      pc=s17:     (pc,struct_p[temp].age):=(s18,0);
      pc=s18:     (re):=(0)&RETURN;
SPEC
    AF(pc=s18);
```

图 3　指针

5.3 函数

函数调用是保证整个项目灵活性的重要部分. 函数调用是许多 C 语言模型检测工具难以解决的部分. 在 Verds 模型中有 procedure 部分的模块, 使我们可以解决这一难题.

对于函数定义, Verds 和 C 都在独立的模块进行的, 这也是函数定义的特点. 函数的声明, 在 Verds 模型中函数没有声明, 函数的参数, 在 Vreds 中只要是可以支持的都可以作为参数, 如果有些参数的操作在 5.1 中语句转化的定义中没有, 我们认为这是不支持的. 函数的返回值, Verds 模型中, 如果有返回值, 那么这个返回值是作为函数的一个参数, 从图 4 中, 可以看到, 返回值作为参数, 在函数体内进行了赋值, 在函数调用阶段进行值的传递, 完成赋值.

Verds 模型可以进行递归函数的验证也是 Verds 模型的一大优势, 图 4 用递归完成了两个数的最大公约数计算, 并给出 Verds 模型.

```
C:
int gcd(int x, int y){
    int temp=0;
    if(y==0) return x;
    else{
            n=gcd(y , x%y);
            return n;
        }
    }

    Verds:
    PROCEDURE gcd(x, y, rt)
    VAR
        temp:  int;
        pc:    {s0,s1,s2,s3};
    INIT
        temp=s0;
        pc=s0;
    TRANS
        pc=s0:      (pc,temp):=(s1,0);
        pc=s1&(y==0):      (rt):=(x)&RETURN;
        pc=s1&!(y==0):     (pc):=(s2);
        pc=s2:      gcd(y,x%y,n)&(pc):=(s3);
        pc=s3:      (rt):=(n)&RETURN;
```

图 4 递归调用

6 结合CEGAR的模型检测

从图 5 所示, 我们看到反例引导的抽象精确框架有四部分, 一是抽象, 二是验证, 三是反例确认, 四是精化. 本问以图 6 来说明. 验证第十条语句不可达性质.

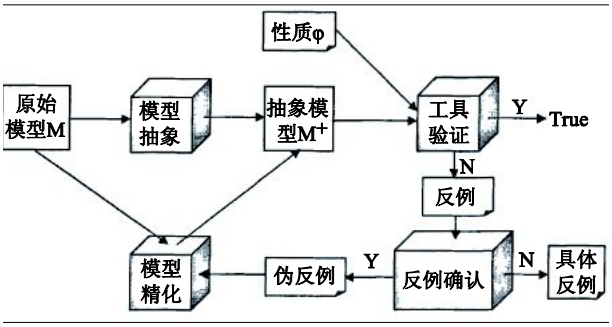

图 5 反例引导的抽象精化框架

```
C:
int i;
int k;
voide get(){

1       int can_enter =0;
2       if(i==0){
3           if(k>10){
4               new();
5               i=1;
6               can_enter=1;
            }
          }else
7           can_enter=1;
8       if(can_enter==1)
9           if (i==0)
10              assert)(False ;
11          else
12              ungot();
    }
```

图 6 CEGAR 例子

6.1 抽象

首先框架第一步是抽象. 采用一定的方法(如谓词抽象)对待检验的模型进行压缩, 使模型的状态空间保持在一个可接受的范围内, 在一定程度上减缓“状态爆炸”的程度.

我们对图 6 中的进行布尔抽象, 图 7 给出了对图 6 中 C 代码进行第一步抽象得到的抽象后的模型, 这是最开始的粗糙的变换. 条件语句我们用“？”表示, 指示不确定的条件限制, 对于语句用“…”来表示没有任何的操作 skip, 图 7 中的代码显示没有任何的状态, 状态最小化. 把程序记为 B_1.

6.2 验证

将抽象后的模型和待验证的性质输入到模型检验工具检验. 如 Clarke 等人采用的就是 SMV 工具. 通过模型检验工具会产生两种结果: 一是模型满足性质,

验证通过并且验证结束; 二是模型不满足性质, 验证未通过, 通常在这种情况下模型检验工具会返回反例.

我们对图 7 中代码和待验证的性质进行验证, 在本文采用将 C 代码和待验证的性质转化成 Verds 模型进行验证, 可以得到在 Verds 工具下的检测结果是性质不正确的, 并给出反例, 将 Verds 模型下的反例路径转化成 C 代码中的路径, 本次反例路径为[1,2,7-10].

```
voide get(){
1       ...;
2       if(?){
3           if(?){
4               ...;
5               ...;
6               ...;
            }
        }else
7           ...;
8       if(?)
9           if(?)
10              ...;
11          else
12              ...;
  }
```

图 7　CEGAR 对图 6 的第一步抽象结果

6.3 反例确认

这一步是框架的一个重点所在. 当第 2 步的工具验证未通过, 返回了反例的时候, 由于该反例可能是一个伪反例, 因此必须确定其真伪性. 通常的作法是构造出该反例在具体模型中所对应的反例(具体反例), 如果在具体模型中存在这样一个具体反例, 则说明具体模型也不满足性质, 验证结束; 如果在具体模型中不存在对应反例, 则说明这是一个伪反例, 需要进入精化.

在 Verds 模型对 B_1 性质验证未通过, 给出该反例, 得到反例路径为[1,2,7-10], 对该路径在具体模型是否也存在反例我们采用 MAGIC 工具产生该路径的 LTS 模型, 并判断是否存在具体反例. 经过 MAGIC 工具验证, 该反例在具体模型中是不存在的. 这说明该路径是一个伪反例, 需要对 B_1 进行精化.

6.4 精化

当抽象模型中伪反例出现的时候, 应该对其进行精化操作. 精化的思想是根据伪反例, 修正抽象模型, 通常是对抽象模型的局部进行重新抽象, 使得精化后的模型不再出现该伪反例.

根据图 7 中路径[1,2,7-10]修正抽象模型, 加入布尔变量 ib 来表示条件判断表达式 i==0, 得到图 8 所示的精化的结果 B_2.

```
boolean ib;
voide get(){
1       ...;
2       if(ib){
3           if(?){
4               ...;
5               ib:=F;
6               ...;
            }
        }else
7           ...;
8       if(?)
9           if(ib)
10              ...;
11          else
12              ...;
  }
```

图 8　CEGAR 对图 7 的第一步精化的结果 B_2

6.5 结果

重复以上过程, 直到找不到反例路径, 则说明性质为不正确, 或者消除不掉反例路径, 则为不确定.

图 9 显示了第二步的精化的结果 B_3, 它完成了模型的抽象验证过程.

```
boolean ib;
voide get(){
1       boolean ce=False;
2       if(ib){
3           if(?){
4               ...;
5               ib=F;
6               ce=True;
            }
        }else
7           ce=True;
8       if(ce)
9           if(ib)
10              ...;
11          else
12              ...;
  }
```

图 9　CEGAR 对图 7 的第二步精化的结果 B_3

7　工具与实验

7.1 工具框架

在将 C 语言子集转化到 Verds 模型的过程中, 运用 lex 与 yacc 来完成对 C 语言的词法分析和语法分析,

并运用 C 语言来传达语法分析的行为，即是转化成 Verds 模型. 如图 10 所示.

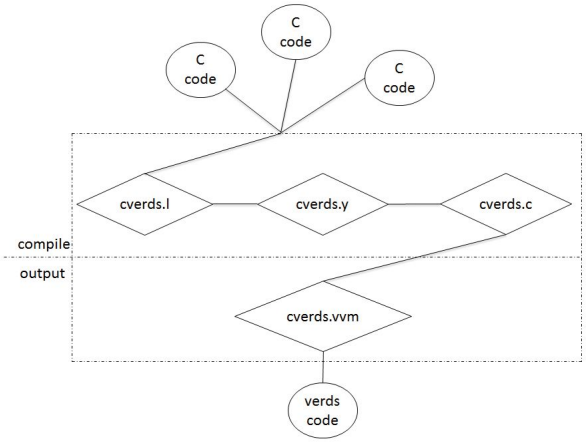

图 10 C 到 Verds 模型转化流程

7.2 实验

本文用四个版本的蓝牙驱动程序中错误作为例子进行验证，驱动程序是并发 C 程序，它有两种类型的线程，分别是停止和加入. 一个停止线程调用停止函数挂起驱动，加入线程调用加入函数来实现驱动中的 I/O. pendingIo 表示并发执行的线程数，没加入一个就加一，离开减一; driverStoppingFlag 表示为真时，线程试图停止驱动; stoppingEvent 表示模拟一个停止事件，初始化为假，当事件发生设置为真. Stopped 初始化为假，但它为真时，驱动安全停止. Contexts 是每个版本产生错误所需要的条件数.

对蓝牙驱动并发程序进行抽象，将之转化为可支持的 C 语言程序，再将 C 程序转化 Verds 模型进行检验. Verds 运行在一台 X86-64 架构的 Linux 服务器上进行了实验. 服务器的配置是 4 路 8 核 Inter Xeon 处理器，每个核为 2.9GHz，300GB 内存，操作系统为 Linux 2.6.18, Murphi 版本为 cmurphi5.4.4.

表 2 实验结果

	Version1	Version2	Version3
Time(s)	11.5	38．0	22.5
Contexts	3	5	4

8 总结与展望

总的来说，本文引入基于 verds 工具的 C 语言子集模型检测研究，实现将 C 语言转化到 Verds 模型，完成对 C 语言子集在 Verds 模型中自动化验证. 为了更好的完成对 C 语言子集的性质验证，将 CEGAR 和 Verds 结合，实现程序的抽象精化，该过程运用辅助工具——MAGIC 来完成. 本文有效实现对 C 语言程序的验证和状态爆炸问题的解决.

本文未来的工作方向有以下四个方面: 一是指针变量命名，现在的版本只能处理 26 个英文字母的指针变量的命名方式; 二是多维数组的处理，现在仅支持一维数组的处理，对于多维数组的处理还需要更多的思考; 三是指针类型，指针是 C 语言的核心，对指针更多类型的处理是必须的，如 viod 指针，指向指针的指针和函数指针; 四是抽象精化，我们提出 CEGAR 的抽象精化，但是它的实现是半自动化的，希望能更加的自动化，完全自动化有些难度.

参考文献

1 林惠民,张文辉.模型检测:理论、方法与应用.电子学报, 2002,30(12A):1907,1912.

2 Clarke E, Grumberg O, Jha S, Lu Y, Weith H. Counterexample-guided abstraction refinement. Proc. of the 1 2m Int'l Conf. on Computer Aided Verification. Berlin, Springer-Verlag. 2000. 154–169.

3 Ball T, Levin V, Rajamani SK. A decade of software model checking with SLAM. Commun. ACM, 2011, 54(7): 68–76.

4 Beyer D, Henzinger TA, Jhala R, Majumdar R. The software model checker blast. STTT, 2007, 9(5-6): 505–525.

5 Blazy S, Leroy X. Mechanized semantics for the clight subset of the C language. J. Autom. Reasoning, 2009, 43(3): 263–288.

6 Chaki S, Clarke E, Groce A, Jha S, Veith H. Modular verification of software components in C. (ICSE). 2003. 385–395.

7 Zhang W. Bounded semantics of CTL[Technical Report]. ISCAS-LCS-16, Institute of Software, Chinese Academy of Sciences. 2010.

8 Zhang W. Ternary boolean diagrams[Technical Report]. ISCAS-LCS-10-24, Institute of Software, Chinese Academy of Sciences. 2010. M. Ma. Model Checking for Protocols Using Verds. TASE 2011: 231–234.

9 Zhang W. VERDS modeling language. Manuscript, SKLCS, 2012. http://lcs.ios.ac.cn/~zwh/verds/.

10 Chaki S, Clarke E, Kidd N, Reps T, Touili T. Verifying

(下转第 18 页)

但由于本文缺乏定量分析, 所以未来的研究需要对企业 IT 决策机制进行定量测量. 此外, 从云计算应用视角出发来研究 IT 治理的其他相关问题, 也应成为未来的研究的重点和方向.

参考文献

1 石菲.云计算:外包 2.0,中国计算机用户,2009(11):12–13.

2 Sambamurthy V, Zmud RW. Arrangement for information technology governance: A theory of multiple contingencies. MIS Quarterly, 23(2): 261–290.

3 Van Grembergen W,De Haes S.Guldentop, structures, processes and relational mechanisms for IT governance. Strategies for Information Technology Governance. London, Idea Group, 2004: 1–36.

4 Weill P, Ross J.A Matrixed approach to designing IT governance. MIT Sloan Management Review WIinter, 2005, 46(2).

5 Olson MH, Chervany NL. The relationship between organizational Characteristics and the Structure of the Information Services Function, MIS Quarterly, 1980, 4(2): 57–68.

6 Mahmood Z. Architectural representations for describing enterprise information and data. Proc. 10th WSEAS Conference on Computers, Athens, Greece. July 2006. 728–733.

7 Davenport TH, Hammer M, Metsisto TJ. How executives sharp their company's information technology systems. Harvard Business Review. 1989, March-April. 130–136.

8 Peter GW. Every manager's guide to information technology. (2nd Edition). Boston, Harvard Business School Press, 1995: 23–44.

9 Ross EW. Creating strategic IT architecture competency: learning in stages. Massachusetts Institute of Technology. 2003.

10 Weill P, Broadbent. Leveraging the new infrastructure: how market leaders capitalize on information technology. Boston, Harvard Business School Press, 1998: 4–10.

11 Weill P, Ross J. IT governance: How top performers manage IT decision rights for superior results. Boston, Harvard Business School Press, 2004: 12–20.

12 Ross J, Beath C. Beyond the business case: strategic IT investmen. Boston, Harvard Business School Press, October 2001.

13 Hayes B, Brunschwiler T, Dill H. Cloud computing. Communications of the ACM, July 2008: 9–11.

14 张嵩,李文立,黄丽华.电子商务环境下企业 IT 基础设施能力的构成研究.计算机集成制造系统,2004(11):10–11.

15 Zachman. The framework for information systems architecture. IBM Syst. 1987, 26(3): 276–292.

16 Mahmood Z, Hill R. Cloud computing for enterprise architectures. Springer Science, 2011.

17 Weill P, Subramani M. Marianne broadbent building IT infrastructure for strategic agility. MIT Sloan Manage- ment Review, Fall, 2002: 57–67.

18 Mahmood Z, Hill R. Cloud Computing for Enterprise Architectures. Springer Science, 2011.

(上接第 25 页)

concurrent message-passing C programs with recursive calls. Proc. TACAS. LNCS 3920. 2006. 334–349.

11 Chaki S, Clarke E, Groce A, Jha S, Veith H. MAGIC: Automated compositional abstraction refinement for concurrent C programs. CMU 2004. http://www.cs.cmu.edu/~chaki/magic/.

12 Chaki S, Clarke E, Groce A, Jha S, Veith H. Modular verification of software components in C. ICSE. 2003. 385–395.

13 Ball T, Majumdar R, Millstein TD, Rajamani SK. Automatic predicate abstraction of C programs. PLDI 2001:203–213.

14 Hashimoto Y, Nakajima S. Modular checking of C programs using SAT-based bounded model checker. APSEC. 2009. 515–522.

15 Ball T, Rajamani SK. A model and process for software analysis[Technical Report]. MSR-TR-2000-14.

16 Jiang K. Model checking C programs by translating C to Promela. Uppsala Universiet, 2009.

17 Li L, Song XY, Gu M, Luo XY. Competent predicate abstraction in model checking. Science China Information Sciences, 2011, 54(2): 258–267.