ucore OS 建模分析工具 –ucore-os-analyser

陈渝　　皮一凡　　林衍凯　　冯齐纬

2013

1 概览

ucore OS 分析建模工具 ucore-os-analyser(简称 UA) 建立在 Commuter [1] 基础上。Commuter 是一个对操作系统接口定义进行符号化描述、验证和生成测试用例的软件工具，其主要实现部分用 python+z3py 完成。在 [1] 中，作者提出名为 Commutativity 的规则：只要接口定义可以交换运行不冲突，就能有可多核扩展的实现。由于 Scalability 不是我们关注的重点，这里不再赘述。

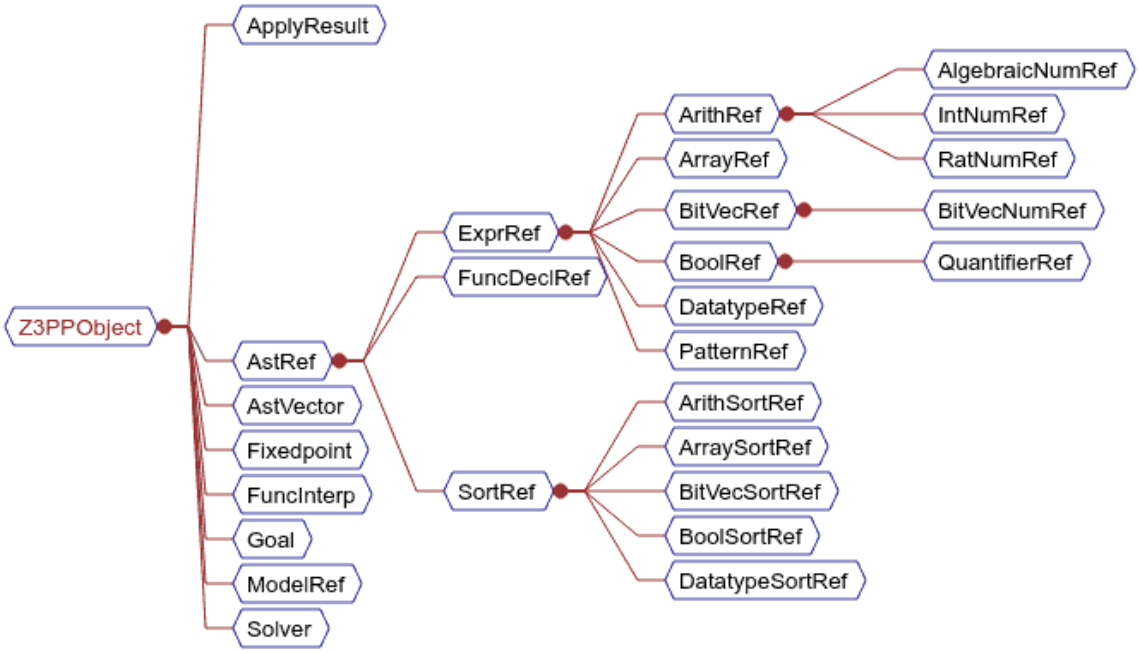
UA 的目标是对 ucore OS 进行抽象建模，并能够分析 ucore OS model 的逻辑行为与逻辑关系，以及 ucore OS model 与 ucore OS 的 C 实现代码之间的一致关系，能够根据 ucore OS model 自动生成测试用例来测试 ucore OS 实现的正确性等。

对于软件的形式化描述，传统的方法基本上是人工进行对实际代码进行抽象后，再手写适对应的专门逻辑代码，这些专门的逻辑代码与传统的命令式语言或脚本类语言有较大差别，学习起来比较困难，如 [3]。而 UA 基于 Python 语言实现了一个形式化描述引擎，使得我们可以使用一种基于流行的 Python 语法，用较为自然的方法来形式化描述系统的接口和功能。如下代码是对于 POSIX 接口规定的 rename 函数的形式化描述。可以看出，这样描述与原始的实现类似，可读性很高。

```
@symargs(src=SymFilename, dst=SymFilename)
def rename(self, src, dst):
        if not self.fname_to_inum.contains(src):
                return (-1, errno.ENOENT)
        if src == dst:
                return 0
        if self.fname_to_inum.contains(dst):
                self.inodes[self.fname_to_inum[dst]].nlink -= 1
        self.fname_to_inum[dst] = self.fname_to_inum[src]
        del self.fname_to_inum[src]
        return 0
```

UA 通过采用 python 的面向对象的基本方法和操作符重载，UA 可以通过自定义的符号执行引擎（Symbolic Execution Engine）分析并遍历执行某函数调用所有的可能执行路径，并将执行路径中存在的逻辑条件（path condition）转换为符号定理证明系统 Z3 [2] 可以识别的逻辑表达式；然后使用 Z3 分析逻辑表达式的合理性（检查是否 sat)；最后通过 Z3 产生出满足 sat 的具体值（及 Z3 的 model)，并根据这些具体值生成相应的 C 语言测试用例，可测试 ucore 操作系统的正确性等。

UA 的核心部分在于设计的各种符号数据类型，符号数据结构以及符号计算引擎，其形式与一个特点的语言解释器的工作方式类似。使用 UA，我们可以对操作系统的各个部分进行形式化描述。下面将主要介绍一下 UA 中符号计算引擎的工作原理和接口实现。



2 UA 简介

UA 的符号计算的重要数据定义和执行的核心实现主要在 *simsym.py* 和 *simtypes.py* 两个文件中，总控程序在 *ucoresymexe.py* 文件中。我们将首先介绍一下 Z3 的基本功能，接着分析 UA 是如何封装 Python 中基本数据类型运算、分支结构和函数调用的。

2.1 对 Z3 的使用

Z3 是一个强大的定理证明引擎，可以完成多种类型的符号计算任务。在 UA 中，主要用到 Z3py（python 对 Z3 的封装包）提供的 Z3 功能是 Model Check，以及对应的 Int，Bool，Array 等基本符号数据结构。即给出一系列逻辑表达式，判断是否可满足，并找出一组可以满足的赋值。下面的例子显示了一个最简单的 Z3 中 Model Check 的用法。

```
x = Int('x') #建立实型的符号变量
y = Int('y')
s = Solver()
s.add(x + y > 5, x > 1, y > 1) #加入逻辑表达式
print(s.check()) #判断是否可满足
print(s.model()) #输出一组满足的赋值
```

输出结果为

```
sat
[y = 4, x = 2]
```

UA 中对 Z3 的使用基本与上个例子类似，通过建立一个 Solver，并根据 UA 对函数的符号遍历执行，在整个执行路径中不断加入逻辑表达式，最后使用 Z3 判断是否满足并产生相应的具体赋值，再根据赋值去生成测试用例。Z3py 封装的各种 z3 符号数据类型引用如下图所示：

2.2 符号数据结构的类层次

Symbolic 是所有符号数据类型的基类。符号类型分为两组，一组具有 constant 属性的符号数据类型，一组是具有 mutable 属性的符号数据类型。具有 constant 属性的符号数据类型的符号变量是 deeply immutable，即与函数语言中的一般数据类型是一致的，只能赋值一次。而具有 mutable 属性的符号数据类型主要用于复合数据结构和 container 类型的数据结构允许对其内部 field（比如 SStructBase）和 element(比如 SMapBase) 进行多次赋值。

```
Symbolic + MetaZ3Wrapper
   |----SymbolicConst
   |----SSynonymBase
   |----SMapBase      #mutable compound datatype
   |----SStructBase   #mutable container datatype
   |----SListBase     #tlist uses tstruct and tmap to create  SListBase based datatype
   |----SDictBase     #tdict uses tstruct and tmap to create  SDictBase based datatype
   |----SSetBase      #tset uses tstruct and tmap to create  SDictBase based datatype
   |----SBagBase      #tbag uses tstruct and tmap to create  SDictBase based datatype
   |----SExpr
          |----SBool
          |----SEmuBase
          |----SUninterpretedBase
          |----STupleBase
          |----SConstMapBase
          |----SArith
                 |----SInt
```

2.3 基本符号数据类型

UA 中的基本符号数据类型主要定义在 *symtypes.py* 和 *simsym.py* 中，UA 中供用户使用的基本符号数据类型只有 *SBool* 和 *SInt*，这两个数据类型支持基本的布尔运算和整数运算，并将结果记录。这里，我们特别说明以下语句。

```
#a is a SInt
a = a + 1
simsym.assume(a >= 2)
```

这里的第一条语句是一个赋值语句，但与传统的赋值并不相同。首先，a 不再是一个数值变量，再执行了赋值语句后，它实际变成了 $a+1$ 这样一个表达式。再者，UA 中最后进行判断的是逻辑表达式，如果没有加入 *simsym.assume* 函数或者进行逻辑判断操作 (及执行 if, while 等包含逻辑判断的语句)，赋值操作是不会反映在 Z3 最后的符号计算和推导中的。如果我们删掉 *assume*，$a+1$ 将不会在最后的逻辑表达式中出现。

通过基本的数据操作，我们可以用简单的 Python 语句描述数据间的相互关系，而不是 Z3 中的 *add* 等特殊操作。上一小节的 $x+y>5 \wedge x>1 \wedge y>1$ 可以用 UA 写为：

```
#x, y, z are SInt
simsym.assume(x > 1)
simsym.assume(y > 1)
z = x + y
simsym.assume(z > 5)
```

可以看出，x, y, z 间的关系简单地描述为 $z=x+y$，比较直观。

值得一提的是，*SBool* 中重载了非零判断，这将用来自动生成分支路径，从而可以遍历所有的执行路径。这将在下文中进行进一步分析。

2.3.1 SInt

SInt 基本类型的简单说明:

```
"""
SInt常用基本符号数据类型
对应的Z3类型  z3_sort__ = z3.IntSort()
构造变量的方法  SInt.var()
使用例子 models/watermark.py
"""
import simsym
import symtypes
import model
class Watermark(simsym.tstruct(model_val=simsym.SInt)):
            @model.methodwrap(param_val=simsym.SInt)
            def put(self, param_val):
                if param_val > self.model_val:
                    self.model_val = param_val

            @model.methodwrap()
            def max(self):
                return self.model_val
model_class = Watermark
```

2.3.2 SBool

SBool 基本类型的简单说明:

```
"""
SBool  常用基本符号数据类型
对应的z3类型 _z3_sort__ = z3.BoolSort()
重载函数 "__eq__", "__ne__", ""__nonzero__"
其中的成员函数__nonzero__ 用来产生新的执行路径
构造变量的方法:  SBool.var()
使用例子 models/state.py
"""
import simsym
import symtypes
import model
class State(simsym.tstruct(ok=simsym.SBool)):
      @model.methodwrap()
      def true(self):
          self.ok = True
      @model.methodwrap()
      def false(self):
          self.ok = False
      @model.methodwrap(ok=simsym.SBool)
      def output(self,ok):
          if (self.ok==ok):
                print "same"
          else:
                print "diff"
model_class = State
```

2.3.3 SUninterpretedBase

SUninterpretedBase 类型的简单说明:

```
等价于一种z3的无约束符号常量类型, 算是一种 SymbolicConst 类型
对应的创建函数 tuninterpreted(name)
例子
SPipeId = simsym.tuninterpreted("SPipeId")
```

2.3.4 equivalent Type

tsynonym 函数的说明：

```
定义一个符号数据类型baseType的别名
定义符号数据类型别名 tsynonym(name, baseType)
例子
SOffset = simsym.tsynonym("SOffset", simsym.SInt)
```

2.4 高级符号数据结构

UA 中的高级数据结构主要定义在 *symtypes.py* 和 *simsym.py* 中，有 *SStructBase*、*SMapBase*、*STupleBase*、*SBagBase*、*SDictBase*、*SListBase*、*SSetBase*、*SEnumBase* 几种。尽管 Z3 中提供了构造队列，树等递归数据结构的描述，但是 UA 中并没有直接采用。由此我们可以发现这些高级数据结构即具有函数式语言特有的符号变量特征，也具有命令式语言的内存变量的特征。这表现在可以对这些变量进行直接的操作，而不需要赋值。另外，这些数据结构中实际保存了别的符号变量，但在操作时会加入逻辑语句判断合法性，之后再翻译为 Z3py 的相关语句。这点我们可以在 *SListBase* 的类实现代码中看到。为了方便构造这些数据类型的变量，UA 提供了对应的封装函数 *tstruct*,*tmap*,*ttuple*,*tbag*,*tdict*,*tlist*,*tset*,*tenum*

除了 *SStructBase* 和 *SEnumBase*，其他各个高级数据结构具体的实现其实都是 *simsym.py* 中 *SMapBase* 的一个封装。而 *SMapBase* 则是对 Z3 元语句 *z3.Select* 和 *z3.Store* 的一个再封装。所以，本质上我们只使用了 Z3 数组的符号化描述，实现了其他的数据结构。这样多层封装的好处是我们可以在根据数据结构的特点，加入前期的逻辑判断，减少对 Z3py 的语句调用。

所以，这些数据结构使用起来和普通的数据结构一样，不同的在于我们可以使用符号变量对其进行操作。SStructBase 和 SMapBase 是两种 mutable 的数据结构，类似 C 语言的 struct 和 array。这里对 mutable 的解释是”Mtable values act like lvalues except when assigned, where they are copied at the point of assignment.”。可以看出，对 SStructBase 中的 field 和 SMapBase 的 element 可以通过赋值语句（assignment）进行修改。这是与函数式语言一般没有赋值语句的特点有一定的区别。

2.4.1 SStructBase

SStructBase 类型的简单说明：

```
是一种mutable的复合符号数据类型
构造函数 tstruct(a=SInt, b=SBool, c=SInt, d=SInt).var("s1")
例子
t1 = tstruct(a=SInt, b=SBool)
t1.a=42
t1.b=True
```

2.4.2 SMapBase

SMapBase 类型的简单说明：

```
是一种mutable的复合符号数据类型
构造函数 (, ) tmapindexTypevalueType
其他函数 constVal(value) #设置Map里所有element的值为value
例子: (, )
tmapindexTypevalueType.constVal(value)
t1 = tstruct(a=SInt, b=SBool)
x = tmap(SInt, t1).var()
```

```
x[0] = t1.var(a=42, b=True)
assert x[0].a == 42
assert x[0].b == True
```

2.4.3 SBagBase

SBagBase 类型的简单说明:

```
是一种能够支持多个相同元素存在的集合
构造函数 tbag(valueType)
成员函数
   add(self,val) #增加一个元素 assume(self._imap[val]>0)
   take(self)  #减少一个元素 assume(self._imap[val]>0)
注意: 不建议使用, 里面有个实现有问题add_internal(v)不存在
```

2.4.4 SDictBase

SDictBase 类型的简单说明:

```
是一种字典
构造函数 tdict(keyType, ) valueType
成员函数
  create(keyType key) #返回值Dict键值为key的值, 如果没有该键, 则新建一个, 返回一个valueType
  contains(keyType key) #返回值 Dict 内是否有键值key
  empty() #返回值Dict是否为空
```

2.4.5 SListBase

SListBase 类型的简单说明:

```
一种以整数为索引下标的列表
构造函数 tlist(valueType, lenType=SInt)
成员函数
  len() #返回此list的长度, 返回值的数据类型是一个Sint类型的符号变量
  shift(SInt by=1) #无返回值, 将list前by个元素删去
  append(valueType val) #在list最后加入元素val
例子利用SListBase和SMapBase统计数字个数
class countnum(simsym.tstruct(countnum1=simsym.tmap(simsym.SInt,simsym.SInt).constVal(0))):
            @model.methodwrap(array1=symtypes.tlist(simsym.SInt,simsym.SInt))
            def __init__(self, array1):
                i = simsym.SInt.var()
                simsym.assume(i==0);
                #if the number is too large, the branch number will be very much
                simsym.assume(array1.len()<30);
                while (i<array1.len()):
                    self.countnum1[array1[i]]=self.countnum1[array1[i]]+1;
                    i = i + 1
            @model.methodwrap(x=simsym.SInt)
            def get(self, x):
                return self.countnum1[x];
```

2.4.6 SSetBase

SSetBase 类型的简单说明:

```
一种不能有重复元素的集合
构造函数 tset(valueType)
成员函数 empty, all, add, clear, discard, contains
```

```
        empty()
            返回值: 集合是否为空
        all():
            返回值: 集合所有的元素为键值的ConstMap, 值为True
        add(val):
            向集合中加入值val
        clear(val):
            清空集合, 这里val可随意指定, 函数没有使用到
        discard(val):
            在集合中删除值val
        contains(val):
            返回值: 集合是否包含val
```

2.4.7 SEnumBase

SEnumBase 类型的简单说明:

```
一种枚举类型, 对应Z3的EnumSort
构造函数 tenum(name, vals)
例子
enum1 = simsym.tenum("enum",["lyk","pyf","fqw"])
```

2.4.8 STupleBase

STupleBase 类型的简单说明:

```
目前不能用
```

2.5 总体控制

UA 的总体控制实现在 *ucoresymexec.py* 中，大致的执行流程为:

```
main--> simple_do_call-->symbolic_apply-->simple_test()-->model funcitons
```

2.6 逻辑操作函数

在写 model 时可用到的逻辑操作函数的说明如下:

2.6.1 symeq

```
symeq(*exprs):
表示列表的相等关系
例子
assume(symeq(a,b,c,d))
表示a==b AND b==c AND c==d
```

2.6.2 symif

```
symif(pred, cons, alt):
表示: 如果pred条件成立, 则返回cons, 否则返回alt
在实现中使用了z3.If
例子:
assume(symif(a, b, c))
```

2.6.3 distinct

```
distinct(*exprlist):
表示: 表达式列表两两不同
在实现中使用了z3.Distinct
例子:
assume(distinct(a,b,c))
表示 a != b AND a!=c AND b!=c
```

2.6.4 implies

```
implies(a, b):
表示 a->b 相当于: (not a) or (b)
例子:
assume(implies(a, b))
表示 a->b
```

2.6.5 exists

```
exists(vars, e, patterns=[]):
表示存在 1 个或多个vars的某个值，满足逻辑表达式e
在实现中使用了z3.Exists
例子:
assume(exists(x, x+y<3))
表示存在x的取值，使得x+y<3成立
```

2.6.6 forall

```
forall(vars, e, patterns=[]):
表示对于所有 1 个或多个vars的取值，都满足逻辑表达式e
在实现中使用了z3.ForAll
例子:
assume(forall(x,x+y>4))
表示对于任意x的取值，x+y>4都成立
```

2.6.7 symor,symand,symnot

```
symor(exprlist)
symand(exprlist)
symnot(e)
例子
    symor(a+b<4,a+b>3)
    symand(a+b<4,a+b>5,b+c<3)
    symnot(a+b<5)
```

2.6.8 assume

```
assume(e)
表示符号表达式 e 为真，且这个表达式将加入 env 环境类的对象和 scheduler 调度类的对象中
作为一个 path condition 的一部分
例子
if x > y:
   simsym.assume(x > z)
```

2.7 辅助函数

一些重要的辅助函数说明:

2.7.1 wrap,unwrap

```
unwrap(val):    把一个 UA 符号值转换成一个 z3 符号值
wrap(val):把一个 z3 符号值转换为一个 UA 符号值
z3 value to Symbolic;
z3.ArithRef/z3.BoolRef value to Symbolic;
```

2.7.2 simplify

```
simplify(expr, try_harder=False)
表示用 z3 化简表达式 expr
```

2.8 分支结构

对于分支结构的自动判断并生成多条路径，是 UA 相比 Z3py 的主要改进。通过前面的分析，可以发现 UA 对于数据类型的改变只是一个再封装，我们仍然可以使用 Z3 的提供的 API 直接描述基本的数据运算。但是，如果出现了分支判断，例如:

```
if expr:
   foo
else
   bar
```

则我们需要手工建立两个 Z3 Model，其中一条为 expr ∧ foo，另一条为 (¬ expr) ∧ bar。如果出现循环的嵌套或者递归，那么手写将变得十分困难。同时，在某些情况下，我们可以根据前文直接判断出 expr 的取值，从而减少分支的数量。如果人工的分析，将很难排除这样的情况。

在 UA 中，通过重载 *SBool* 的非零判断成员函数 ___nonzero___，UA 首先判断逻辑表达式可能的取值。如果取值只有一种，则直接顺序执行。如果取值有两种，那么则在执行队列的末尾加入两种情况，分别执行。其执行方式类似一种宽度优先搜索。考虑下面语句

```
@model.methodwrap()
def do(self):
        if(self.counter > self.counter1):
                self.counter = self.counter - 1
                simsym.assume(self.counter > 0)
        else:
                self.counter1 = self.counter1 + 1
                simsym.assume(self.counter1 > 1)
```

UA 将自动生成两条执行路径:

```
path is  2
  1 -- And(Counter.counter > Counter.counter1, Counter.counter - 1 > 0)
  2 -- And(Not(Counter.counter > Counter.counter1), Counter.counter1 + 1 > 1)
```

2.9 Model 编写

model 是一个继承 SStructBase 类数据结构的子类，例如 class model(tstruct(a=simsym.Sint))，modle 的成员函数中的所有的控制逻辑由 python 的控制语句完成，但由于 UA 中的各种符号变量的内置成员函数大部分都进行了重载，使得在对这些变量进行各种操作（读，写，比较，选取某个 item 等）时，实际上被转换成了 z3py 支持的各种 z3 操作。特别是 UA 对 SBool 的非零判断成员函数进行了重载，但执行 if 或 while 等判断语句时，将会执行 SBool 的非零判断成员函数，并给 UA 的符号执行引擎增加新的执行路径。而 model 的一个成员函数执行完毕后，UA 会收集此函数最后运行的输出结果是一个 SymbolicApplyResult（简称 SAR），SAR 里的主要信息是这个成员函数一条执行路径的运行条件（path condition）h 和返回值。UA 的符号执行引擎会收集此函数的所有执行路径的 SAR，形成一个 SAR 集合。

model 有一个初始环境（由一些符号成员变量组成）和初始条件（这些符号成员变量的一些逻辑表达式），并包含了各种成员函数（接收某些外部符号变量输入，用来对 model 的内部环境进行逻辑改变，并产生输出）。model 的初始环境所包含的各种符号变量就是它所继承的某个 SStructBase 类的类型的各个 field。比如：

```
class Counter(simsym.tstruct(counter=simsym.SInt, counter1=simsym.SInt)):
```

表示了 Counter model 的初始环境包含两个符号整型变量 counter 和 counter1。

_declare_assumptions 是一个给 model 的初始环境设置初始逻辑条件的成员函数（继承与 tstruct 或者 tmap），在其他成员函数的每次运行前都会调用此函数。你可以写多个辅助的 class，文件的最后必须定义 model_class 是哪一个 class。下面是一个比较完整的小例子：

```
class Counter(simsym.tstruct(counter=simsym.SInt, counter1=simsym.SInt)):
    def _declare_assumptions(self, assume):
        print "declare"
        super(Counter, self)._declare_assumptions(assume)
        assume(self.counter >= 0)
        assume(self.counter1 >= 0)

    @model.methodwrap()
    def inc(self):
        if (self.counter > self.counter1):
            self.counter = self.counter - 1
        else:
            self.counter1 = self.counter1 + 1
model_class = Counter
```

这个 model 的输出是：

```
path is  2
      1 -- And(Counter.counter >= 0,Counter.counter1 >= 0,Counter.counter > Counter.counter1)
      2 -- And(Counter.counter >= 0,Counter.counter1 >= 0,Not(Counter.counter > Counter.counter1))
```

其中 if 语句被通过条件判段发现可以有两条路。在看下面的例子：

```
class Counter(simsym.tstruct(counter=simsym.SInt, counter1=simsym.SInt)):
    def _declare_assumptions(self, assume):
        print "declare"
        super(Counter, self)._declare_assumptions(assume)
        assume(self.counter >= 0)
        assume(self.counter1 < 0)

    @model.methodwrap()
    def inc(self):
        if (self.counter > self.counter1):
            self.counter = self.counter - 1
```

```
12          else:
13              self.counter1 = self.counter1 + 1
14      model_class = Counter
```

我们在保证了 self.counter > self.counter1，所以 if 语句的两条路只有一条是满足的。

```
1   path is  1
2       1 -- And(Counter.counter >= 0,Counter.counter1 < 0, Counter.counter > Counter.counter1)
```

3 实验任务

UA 提供了一个直观简便的方法来描述一个软件系统的模型，我们通过分析 UA 的符号计算引擎基本工作方式等了解了编写一个 Model 的基本思路。可以参考 Commuter 提供已有的 POSIX FS Interface Model 的描述，编写 ucore OS 的在 System Call Interface，Memory Management，File Management, Process Management 的 Kernel Service Interface 的形式化描述，并利用 UA 进行进一步的自动化分析与验证。

4 致谢

UA 工具建立在 MIT 的 Commuter 之上，用于建模和分析 ucore OS。感谢 MIT PDOS Group 的 Professor M. Frans Kaashoek, Professor Nickolai Zeldovich, Professor Robert Morris 和 Ph.d Students Austin Clements, Xi Wang,Yandong Mao, Haogang Chen 等给予的大力帮助。

参考文献

[1] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP 2013)*, Farmington, Pennsylvania, November 2013.

[2] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[3] Nicolas Marti, Reynald Affeldt, and Akinori Yonezawa. Formal verification of the heap manager of an operating system using separation logic. In *Proceedings of the 8th international conference on Formal Methods and Software Engineering*, ICFEM'06, pages 400–419, Berlin, Heidelberg, 2006. Springer-Verlag.

5 附录

5.1 commuter 的 Fs 简介

commuter 设计了 Fs，用于 commutativity 的分析和验证。我们这里只关注其 Fs 的设计思路。

5.1.1 Fs 的环境和环境初始条件

Fs 的环境可根据其 class Fs 所继承的类获知:

```
class Fs(simsym.tstruct(
        i_map=SIMap, proc0=SProc, proc1=SProc, pipes=SPipeMap,
        ## XXX Non-directories impl:
        root_dir=SDirMap)):
```

这里可以看到, i_map 表示 inode 的集合，表明了 Fs model 要描述的文件 inode 集合，这实际代表了文件系统中的所有 inode 集合。proc0 和 proc1 表示操作系统中存在的两个进程。pipes 表示以 SPipeID 为索引的一个列表，代表了创建的 pipe。如果不关心 pipe 系统调用，这里可以暂时不用深入了解。root_dir 是一个一 filename 为索引的 inode num 为值的字典，这里实际代表的是一个根目录结构。

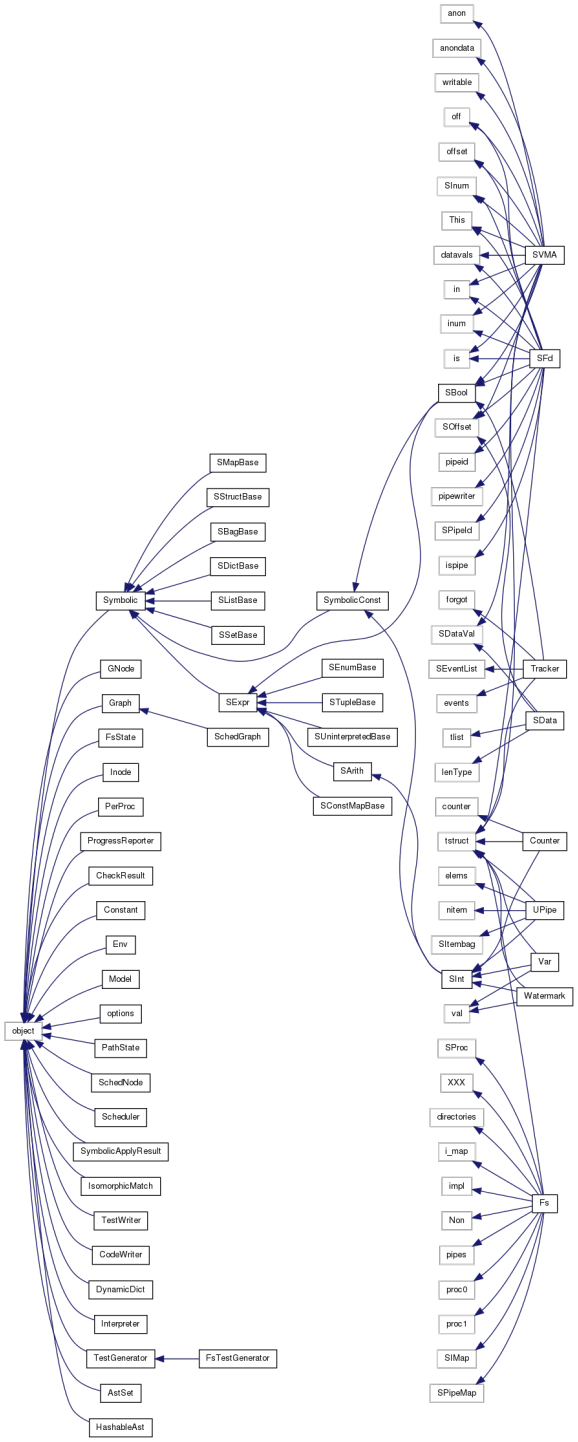
由于 Fs model 没有实现成员函数 _declare_assumptions, 所以其环境初始条件就是 SIMap,SProc, SPipeMap,SDirMap 这些类的 _declare_assumptions 调用的 assume 函数所设置的逻辑条件。

5.1.2 Fs 中的成员函数

从各个成员函数的实现代码，可以大致了解其设计思路。但由于函数中的操作对象都是各种基本数据结构和复杂数据结构的符号变量，导致在具体执行过程中动态生成的执行路径和 path condition 都比较复杂。所以还需仔细理解每个函数的实现细节。

5.1.3 Fs 涉及的部分类层次图

便于大家理解 Fs model 的设计。


anon
anondata
writable
off
offset
SInum
This
datavals
SVMA
in
inum
is
SFd
SBool
SOffset
pipeId
pipewriter
SPipeId
ispipe
forgot
SDataVal
SEventList
Tracker
events
SData
tlist
lenType
counter
tstruct
Counter
elems
nitem
UPipe
SItembag
Var
SInt
Watermark
val
SProc
XXX
directories
i_map
impl
Fs
Non
pipes
proc0
proc1
SIMap
SPipeMap
SMapBase
SStructBase
SBagBase
SDictBase
SListBase
SSetBase
Symbolic
SymbolicConst
SEnumBase
STupleBase
SExpr
SUninterpretedBase
SArith
SConstMapBase
GNode
Graph
SchedGraph
FsState
Inode
PerProc
ProgressReporter
CheckResult
Constant
Env
Model
options
object
PathState
SchedNode
Scheduler
SymbolicApplyResult
IsomorphicMatch
TestWriter
CodeWriter
DynamicDict
Interpreter
TestGenerator
FsTestGenerator
AstSet
HashableAst