符号执行研究综述

叶志斌 严 波
(江南计算技术研究所 江苏 无锡 214083)

摘 要 符号执行作为一种重要的程序分析方法,可以为程序测试提供高覆盖率的测试用例,以触发深层的程序错误。首先,介绍了经典符号执行方法的原理;然后,阐述了基于符号执行发展形成的混合测试、执行生成测试和选择性符号执行方法,同时,对制约符号执行方法在程序分析中的主要因素进行了分析,并讨论了缓解这些问题和提高符号执行可行性的主要方法;随后,介绍了当前主流的符号执行分析工具,并比较分析了其优缺点;最后,总结并讨论了符号执行的未来发展方向。

关键词 符号执行,混合测试,执行生成测试,路径爆炸,约束求解
中图法分类号 TP311 **文献标识码** A

Survey of Symbolic Execution

YE Zhi-bin YAN Bo
(Jiangnan Institute of Computing Technology, Wuxi, Jiangsu 214083, China)

Abstract As an important program analysis method, symbolic execution can generate high coverage tests to trigger deeper vulnerabilities. This paper firstly introduced the principle of classical symbolic execution, and elaborated three dynamic symbolic execution methods which are known as concolic testing, execution generated test and selective symbolic execution. Meanwhile, the essence of main challenges of symbolic execution and the current major solutions were discussed. Symbolic execution has been incubated in dozens of tools which were described and compared in this paper. Finally, the develop directions of symbolic execution were forecasted.

Keywords Symbolic execution, Concolic testing, Execution generated test, Path explosion, Constraint solving

1 引言

符号执行作为一种重要的形式化方法和软件分析技术,采用抽象符号代替程序变量,程序计算的输出被表示为输入符号值的函数,根据程序的语义,遍历程序的执行空间。其在软件测试和程序验证中发挥着重要作用,并可以应用于程序漏洞和脆弱性的检测中[1]。

符号执行的思想由 King 等[2]于 1975 年首次提出,并被应用于程序分析领域。然而经历过短暂的研究高潮后[3-4],符号执行的研究就陷入了低谷。其原因在于,当时程序分析等领域追崇分析方法的完备性,而作为路径敏感分析的符号执行,无法对程序路径穷举遍历,自然不能保证分析的完备性。随着现代计算机计算能力和存储能力的增强,符号执行通过对搜索策略、内存模型的优化以及 Z3 等 SMT 技术的改进,提高了约束求解能力,从而重新获得了研究人员的重视。在过去十几年的研究发展中,涌现了许多符号执行分析框架。符号执行经过了传统符号执行→动态符号执行→选择性符号执行的发展过程,动态符号执行包括混合测试[5-7]和执行生成测试[9-10]两种。这些符号执行技术的原理将在后续章节阐述。符号执行的优势是能够以尽可能少的测试用例集达到高测试覆盖率,从而挖掘出复杂软件程序的深层错误。然而,在实际的软件测试应用过程中,不可避免地会遇到许多限制条件,如路径爆炸问题、约束求解困难集内存建模等问题,这会导致符号执行难以在现实的软件测试应用中达到理想的效果[1,11]。

2 经典符号执行

经典符号执行的核心思想是通过使用符号值来代替具体值作为程序输入,并用符号表达式来表示与符号值相关的程序变量的值。在遇到程序分支指令时,程序的执行也相应地搜索每个分支,分支条件被加入到符号执行保存的程序状态的 π 中,π 表示当前路径的约束条件。在收集了路径约束条件之后,使用约束求解器来验证约束的可解性,以确定该路径是否可达。若该路径约束可解,则说明该路径是可达的;反之,则说明该路径不可达,结束对该路径的分析。

以图 1 中的示例代码为例来阐述符号执行的原理,程序第 9 行存在错误,我们的目标是要找到合适的测试用例来触发该错误。若使用随机生成测试用例对程序实行具体测试的方法,对于整型输入变量 x,y,z 而言,其取值分别有 2^{32} 种,通过随机生成 x,y,z 取值作为程序测试的输入,则能够触发程序错误的可能性较小。而符号执行的处理是,使用符号值代替具体值,在符号执行的过程中,符号执行引擎始终保持一个

本文受国家自然科学基金项目(91430214)资助。
叶志斌(1992-),男,硕士生,主要研究方向为信息安全,E-mail:SOSCapture@163.com(通信作者);严 波(1978-),男,硕士,高级工程师,主要研究方向为信息安全。

状态信息，这个状态信息表示为(pc,π,σ)，其中：

1)pc 指向需要处理的下一条程序语句，其可以是赋值语句、条件分支语句或者是跳转语句。

2)π 指代路径约束信息，表示为执行到程序特定语句需要经过的条件分支，以及各分支处关于符号值 α_i 的表达式。在分析的过程中，将其初始定义为 π=true。

3)σ 表示与程序变量相关的符号值集，包括含有具体值和符号值 α_i 的表达式。

```
1. foo(int x,int y,int z){
2.    a=0,b=0,c=0
3.    if(x>0){a=-2;}
4.    if(y<5){
5.       if(y+z>0){b=1;}
6.       c=2;
7.    }
8.    if(a+b+c==3)
9.          //some error
10.   exit()
11.}
```

图 1　符号执行的示例代码

符号执行算法具体如算法 1 所示。首先，由于 x,y,z 是程序的输入，将 x,y,z 的值定义为符号变量 σ:α,β,γ，且由于还未执行到任何的条件分支，因此初始状态为 σ:{$x\rightarrow\alpha$,$y\rightarrow\beta$,$z\rightarrow\gamma$};π=true。这里需要说明，除去初始被定义为符号变量的 x,y,z 之外，假如在代码运行过程中出现了与 x,y,z 相关的赋值操作，则也须将新产生的变量当成符号变量进行处理。例如：假设在第 2 行和第 3 行代码之间添加一个赋值操作，即令 $w=2*x$，则对应的 w 也会被认为是符号变量，因此程序状态将变为 σ:{$x\rightarrow\alpha$,$y\rightarrow\beta$,$z\rightarrow\gamma$,$w\rightarrow2*\alpha$};π=true。当程序分析到第一个分支判断语句 if $x>0$ 时，将会分叉出两条路径，即 true 路径和 false 路径，从该分支处延伸出两个分析状态，分别是 if $x>0\rightarrow$ σ:{$x\rightarrow\alpha$,$y\rightarrow\beta$,$z\rightarrow\gamma$};π:$\alpha>0$ 和 if $x\leqslant0\rightarrow${$x\rightarrow\alpha$,$y\rightarrow\beta$,$z\rightarrow\gamma$};π:$\alpha\leqslant0$。首先，沿着 true 路径继续搜索，即满足 $x>0$，将遇到第二个分支判断 if $y<5$，从该分支处将再产生两条路径，路径状态分别为 if $y<5\rightarrow\sigma$:{$x\rightarrow\alpha$,$y\rightarrow\beta$,$z\rightarrow\gamma$};π:$\alpha>0\cap\beta<5$ 和 if $y\geqslant5\rightarrow\sigma$:{$x\rightarrow\alpha$,$y\rightarrow\beta$,$z\rightarrow\gamma$};$\pi$:$\alpha\leqslant0\cap\beta\geqslant5$，记录 false 的路径状态；继续沿 true 路径搜索，遇到第三个分支判断 if $y+z>0$，产生两条路径及两个新的状态 if $y+z>0\rightarrow\sigma$:{$x\rightarrow\alpha$,$y\rightarrow\beta$,$z\rightarrow\gamma$};π:{$\alpha>0\cap\beta<5\cap y+z>0$}和 if $y+z\leqslant0\rightarrow\sigma$:{$x\rightarrow\alpha$,$y\rightarrow\beta$,$z\rightarrow\gamma$};$\pi$:{$\alpha>0\cap\beta<5\cap y+z\leqslant0$}。至此，与符号变量相关的操作都已处理完毕，则需要使用 SMT solver 来求解满足各路径约束条件的解，求解得到的解即为沿该路径执行的测试用例。例如，最终执行到 if $y+z>0$ 的路径的解需满足：$\alpha>0\cap\beta<5\cap y+z>0$，得到的解之一可能是 $\alpha=1,\beta=2,\gamma=1$，该测试用例执行程序得到的结果为 $a=-2,b=1,c=2$，执行完该路径后，符号执行继续沿 false 路径 if $y+z\leqslant0$ 进行求解测试，约束条件为 $\alpha>0\cap\beta<5\cap y+z\leqslant0$，得到的解可能是 $\alpha=1,\beta=2,\gamma=-3$，得到结果为 $a=-2,b=0,c=2$。依次类推，执行完第三个分支的两条路径后，将从保存的状态中取出第二个分支处记录的状态，并求解生成测试用例来测试程序。传统符号执行在原理上是可以对程序路径进行全覆盖的，而且可以针对每一路径都生成符合该路径的测试用例。

算法 1　符号执行算法

```
Symbolic execution():
1. Create initial task
2. pc=0,π=∅,σ=∅
3. add task (pc,π,σ) onto worklist
4. while(worklist is not empty):
5.    pull some task (pc,π,σ) from worklist
6.    if it potentially forks at(pc,π,σ)
7.    Execute
8.       if π0⋂p feasible:
9.          add task (pc1,(π0⋂p),σ0)
10.      if π0⋂¬p feasible:
11.         add task (pc1,(π0⋂¬p),σ0)
12.. end while
```

示例代码的程序执行树如图 2 所示，程序中有 3 个分支判断点，总共有 6 条路径，即符号执行引擎需要进行 6 次约束求解，并得到针对 6 条路径的测试用例。其中，求解约束 $x\leqslant0\cap y<5\cap y+z>0$ 得到的测试用例，执行结果为 $a=0,b=1,c=2$，将触发程序错误。

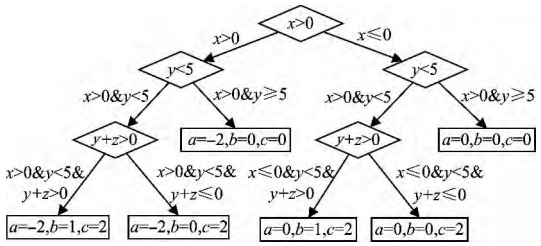

图 2　示例代码的程序执行树

3　符号执行的发展

根据发展状况，可以将符号执行分为经典符号执行、动态符号执行和选择性符号执行。经典符号执行并不真实地执行，而是基于解析程序，通过符号值模拟执行；动态符号执行结合使用了具体执行和符号执行，综合了具体执行和经典符号执行的优点，并出现了混合执行(concolic execution)[5]和执行生成测试[9-10]两种符号执行技术；选择性符号执行可以对程序员感兴趣的部分进行符号执行，其他部分使用真实值执行，代表工具为 S2E[12]。混合执行由 Godefroid 和 Sen 等在 2005 年提出，Sen[8]阐述了其原理，并总结了近十年来的改进与发展；执行生成测试是由 Cadar 等于 2006 年提出的，并在 EXE 和 KLEE 工具中得到了实现；选择性符号执行由 Chipounov 等于 2009 年提出，并在 S2E 框架中得到了实现。

3.1　混合执行

混合执行结合使用具体执行和符号执行的软件测试技术，是动态符号执行的一种。其主要思想是用具体输入执行程序，在程序运行的过程中，通过程序插桩手段收集路径约束条件，按顺序搜索程序路径，利用约束求解器求解上一执行中收集到的约束集，从而得到下一次执行的测试用例；在下一次执行结束后，按一定的策略选择其中某一分支判断点进行约束取反，得到新的约束集，再用约束求解器对其进行求解，得到下一执行的测试用例。如此反复，可以避免执行重复路径，从而以尽可能少的测试集达到高测试覆盖的目的。

以图 1 中给出的示例代码为例，解释混合测试的测试流程。图 3 给出了该程序的路径约束树，可以看出，该程序共有

6 条不同的路径，对于每一条路径，都有其对应的约束集。

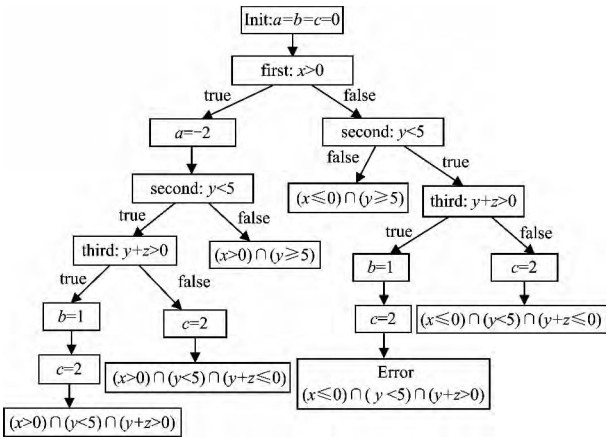

图 3 示例代码的路径约束树

从路径约束树可以看出，该代码有 5 条正常执行结束的路径和 1 条错误路径。通过约束集确定程序执行的轨迹，引导程序沿设定的路径执行。对于示例程序而言，能够触发程序错误的约束集为$(x\leqslant 0)\&(y<5)\&(y+z>0)$，在程序执行过程中，收集并保存该执行路径上的约束。为了对该代码路径进行全覆盖，程序将被执行 6 次，并且每次执行都是通过选取一个约束条件进行取反后再求解出新的测试用例，以测试另一路径。

例如，选取初始测试输入设定 $x=y=z=1$，在具体执行过程中，由于 $1>0$，因此将执行 $a=-2$；接着由于 $1<5$ 且 $1+1>0$，将执行 $b=1,c=2$。程序正常执行结束，在执行过程中收集该路径的约束为$(x>0)\&(y<5)\&(y+z>0)$。为了使下一次执行能覆盖到程序的不同路径，混合测试以一定的策略选择其中的一项分支判定条件进行取反，即将上一执行中收集的约束条件取反，得到新的约束集$(x>0)\cap(y<5)\cap(y+z\leqslant 0)$，通过求解该约束得到新的测试用例 $x=1,y=1,z=-2$；用新生成的测试用例执行程序，执行过程依次为 $a=-2,c=2$，执行结果为$a=-2,b=1,c=2$，程序正常执行结束。在该轮运行之后，没有出现新的约束，因此约束集为$(x>0)\cap(y<5)\cap(y+z\leqslant 0)$，下一轮的约束取反将约束集变为$(x>0)\cap(y\geqslant 5)$，求解生成新的测试用例为 $x=1,y=5,z=-2$，程序依然正常运行结束。依照此过程，反复进行具体执行并收集路径约束，以及约束取反生成新的测试用例的过程，在求解约束集$(x\leqslant 0)\cap(y<5)\cap(y+z>0)$，得到测试用例 $x=-1,y=1,z=2$ 时，执行结果为 $a=0,b=1,c=2$，因为 $a+b+c=3$，所以将触发程序第 9 行的错误。

利用混合测试验证程序的正确性，在理论上是可以对程序路径进行全覆盖的，但是随着分支的增加，程序状态空间呈指数型增长，再加上约束求解的限制，混合测试在现实软件测试中的应用还存在一些问题。例如，Rupak 等提出的 hybrid concolic testing[13] 的核心思想是结合随机测试和混合测试方法对程序进行测试，以保证对程序状态空间搜索的广度和深度。该方法融合了混合测试详尽彻底和随机测试快速高效的优点，提高了程序测试的覆盖率。

3.2 执行生成测试

执行生成测试也是融合了具体执行与符号执行的软件测试技术，由斯坦福大学 Cristian 等提出并在 EXE 中实现。执行生成测试与混合测试最大的不同点在于将符号执行和具体执行混合的方式不同。执行生成测试的混合是在一次程序执行中，对符号变量无关的代码段使用具体执行；而对符号变量相关的代码段进行符号执行，使用符号执行引导测试过程，为每条路径生成一个测试用例，并进行一次执行。

执行生成测试的核心思想是通过程序代码自动地生成潜在的高度复杂的测试用例，在用符号输入执行程序的过程中，在分支处将 false 路径的状态信息记录下来，判断为 true 的分支继续执行。记录其约束信息，通过求解这些约束信息得到该路径的测试用例，该分析过程就是执行生成测试。

下面以图 1 中的示例代码为例来阐述执行生成测试的处理流程。

1)设置初始状态：将 x,y,z 设置为符号变量，且任意取值。由于 x,y,z 的类型为整型，可以将其取值限定在 INT_MIN 和 INT_MAX 之间，因此添加约束条件：$x,y,z\geqslant INT\_MIN\cap x,y,z\leqslant INT\_MAX$，用符号输入执行程序。

2)在第一个条件分支即第 3 行处分叉执行，将 true 分支上的约束条件置为 $x>0$，false 分支上的 x 约束条件为 $x\leqslant 0$。选择其中的 true 分支继续执行，talse 分支以栈的形式克隆存储。

3)在第二个条件分支(第 4 行)处分叉执行，将 true 分支上的约束设置为 $x>0\cap y<5$，false 分支上的 x 约束条件为 $x>0\cap y\geqslant 5$。选择其中的 true 分支继续执行，false 分支克隆存储。

4)在第三个条件分支(第 5 行)处分叉执行，将 true 分支上的约束设置为 $x>0\cap y<5\cap y+z>0$，将 false 分支上约束设置为：$x>0\cap y<5\cap y+z\leqslant 0$。选择其中的 true 分支继续执行，false 分支克隆存储。

5)true 分支由于不满足第 7 行的约束 $a+b+c=3$，执行到程序退出(第 10 行)，求解符号约束 $x>0\cap y<5\cap y+z>0$，得到第一个测试用例。

6)true 分支执行结束，从克隆存储中依次取出分支记录，第一个取出的是约束条件为 $x>0\cap y<5\cap y+z>0$ 的分支，求解得到第二个测试用例。

7)依次从克隆备份中取出分支记录，并求解得到测试用例，直到分支记录栈中为空，得到针对所有路径的测试用例。其中有一分支会执行到错误代码处(第 9 行)，此时的分支约束为 $x\leqslant 0\cap y<5\cap y+z>0$，求解该约束得到触发该错误的测试用例。

与混合测试相比，执行生成测试的优势是能更加系统且高效地得到所有的路径信息以及对应的测试用例，避免重复性搜索过程；其缺点是内存空间耗费较大，一种解决思路是可以使用多线程的方式代替分支存储，实现对多个分支的同时搜索和测试用例的生成。

3.3 选择性符号执行

受路径爆炸和约束求解问题的制约，符号执行不适用于程序规模较大或逻辑复杂的情况，并且对于与外部执行环境交互较多的程序尚无很好的解决方法。选择性符号执行[12,14]就是为解决这类问题而出现的符号执行分析技术，其也是具体执行和符号执行混合的一种分析技术，依据特定的分析，决定符号执行和具体执行的切换使用。在选择性符号执行中，用户可以指定一个完整系统中的任意感兴趣部分进行符号执行分析，可以是应用程序、库文件、系统内核和设备驱动程序。选择性符号执行在符号执行和具体执行间无缝地

来回转换，并透明地转换符号状态和具体状态。选择性符号执行极大地提高了符号执行在实际应用中对大型软件分析测试的可用性，且不再需要对这些环境进行模拟建模。

选择性符号执行在指定区域内的符号化搜索，就是完全的符号执行，在该区域之外均使用具体执行完成。选择性符号执行的主要任务就是在分析前将大程序区分为具体执行区域和符号执行区域。这种选择性是指只对有必要的区域进行符号执行，是将实际应用系统缩放到符号执行可用规模的关键要素。

同样地，以图 1 中的示例代码为例来阐述选择性符号执行的原理，假设仅对代码中第 4－7 行的代码段进行符号分析，而对其余部分进行具体执行。选择性符号执行的核心是符号执行和具体执行的交互处理，即在具体执行转入符号执行区域及符号执行转入具体执行时的处理。

对选定的代码段进行符号分析，其符号变量仅有 y 和 z，而 x 只需作为具体值。假设随机生成的初始输入为 $x=0$，$y=0$，$z=0$，执行程序将得到结果 $a=0$，$b=0$，$c=2$。在代码执行到符号执行区域时，将进行 $0\rightarrow y$ 和 $0\rightarrow z$ 的变换，并进行符号分析。该代码区域的程序执行树如图 4 所示。在该代码区域内的符号分析与全程序的符号分析过程一致，可以根据不同的约束信息求解生成不同的测试用例来对目标程序进行执行，在执行完第 7 行代码后，将符号变量 y 和 z 变换为具体值，即 $y\rightarrow 0$ 和 $z\rightarrow 0$，继续具体执行剩余代码，则本次执行完成。之后，依据符号分析的结果，随机对 x 取值，生成测试用例，如 $x=2$，$y=6$，$z=0$，执行程序结果为 $a=-2$，$b=0$，$c=0$，未触发程序错误，并继续生成测试用，例如 $x=-1$，$y=2$，$z=-3$，执行程序，依据符号执行区域内的分支约束求解执行不同路径的测试用例，直到将目标符号分析区域的路径都执行完毕。由于 x 的取值始终是随机的，因此可能导致即使遍历了符号执行区域内的所有路径，最终也无法触发程序错误。唯有当符号执行区域执行到路径 $y<5\&y+z>0$ 时，x 的取值刚好满足 $x<0$，则会触发程序错误，即程序错误触发的情况仅以一定的概率发生。

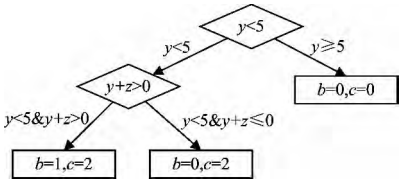

图 4　符号执行区域程序执行树

选择性符号执行的关键挑战在于使这种将符号方式和具体方式表示的数据与执行混合，同时须兼顾到分析的正确性和高效性。因此，需要在具体区域和符号区域设置明确的界限，并且数据必须能够在执行越过所设置的区域界限时，完成符号值域具体值间的转换，这也是选择性符号执行的贡献所在：正确执行一个真实系统及其必要的状态转换任务，在一定程度上达到了最大化本地执行的目的。

4　主要挑战和解决方法

4.1　路径爆炸

路径爆炸问题是制约符号执行在现实程序分析中应用的主要因素。因为在符号执行的分析过程中，在每个分支节点，符号执行都会衍生出两个符号执行实例，程序分支路径的数量与程序分支的数量呈指数级增长关系。以图 5 中的代码为例，代码中共有 3 个分支判断语句和 1 个循环语句，3 个分支判断语句根据 x，y，z 值的不同，将会产生 8 条不同的程序路径；而对于循环来说，当 a 取最大值时，该循环将会产生 2^{31} 条路径。

```
void process(char x,char y,char z,int a) {
  int counter=0;
  if (x=='a') counter++;
  if (y=='b') counter++;
  if (z=='c') counter++;
  while(a--);
      do something(counter++)
}
```

图 5　路径爆炸的示例代码

缓解路径爆炸问题的思路主要有以下几种。

1）采用启发式搜索方法对程序路径空间进行搜索。深度优先搜索 DFS 和广度优先搜索 BFS 是传统的搜索算法，但传统的搜索策略在对大程序进行搜索时表现较为乏力。为缓解路径爆炸问题，Cadar 等[17]在 KLEE 中使用了随机路径选择搜索和覆盖率优化搜索策略（Best-First），在实现多种启发式搜索策略的基础上，KLEE 综合使用了多种启发式搜索策略，以避免单一的启发式搜索陷入局部最大测试覆盖率。UC Berkeley 的 Burnim 等[18]在 CREST 中实现了多种启发式搜索策略，如随机分支搜索和控制流导向搜索策略。Koushik 等[19]在 CUTE 和 jCUTE 中使用了混合随机符号搜索策略，以提高测试的广度与深度。启发式搜索策略的应用能避免搜索陷入饥饿状态，即搜索仅局限在一个小区域内，能以较大的概率执行到未覆盖代码；面对的挑战是其在面临长路径搜索时，对路径的计算和筛选需要耗费较长时间，并且可能无法得到符合的路径。

2）冗余路径剪枝。在程序分析过程中有些路径是冗余的，通过分析确定出冗余路径，并对其进行剪枝。Peter 等[20]提出了判定一条路径是否为冗余路径的方法：①如果两条路径到达同一程序点，并且到达该程序点的约束集相同，则可以对其中一条进行剪枝；②从每条路径的约束集中删除那些与内存相关但再也不会进行读取的约束；③对路径可达性进行预判断，即对每条路径的约束信息进行可满足性判定，如果该约束是不可满足的，则意味着不存在能驱动该路径具体执行的测试用例，该路径为冗余路径，可以直接进行剪枝。冗余路径剪枝可以简化分析的路径空间，从而提高分析效率；而面临的挑战是冗余路径的判定较为复杂，难以全面地对其进行判定，并且对冗余路径的误判可能导致最终分析不到目标。

3）状态合并。Godefroid 等[21-22]阐述了静态状态合并的原理及存在的问题。冗余状态合并虽然能有效地减少待搜索的路径数量，但同时也给约束求解器增加了负担，在进行约束处理解析时容易遇到问题，并且状态合并还有可能引入新的符号表达式。2012 年，Kuznetsov 等[23]提出了一种自动选择何时以及如何进行状态合并的方法，从而显著提高了符号执行的性能。2014 年，卡内基梅隆大学的 Thanassis 等[24]提出了 Veritesting 的概念（Veritesting 即状态拟合）通过状态拟合减小程序的状态空间，提高动态符号执行的可用性。图 6 给出程序状态合并的示意图。状态合并能极大地缩减程序的状态空间，从而减小路径空间，如图 6 所示，其将 2^N 条路径缩减

为 1 条，效果十分显著。但是，状态合并需要满足合并不能造成副作用的条件，即进行状态合并不影响程序分析的准确性，这是状态合并的关键也是其难点所在。

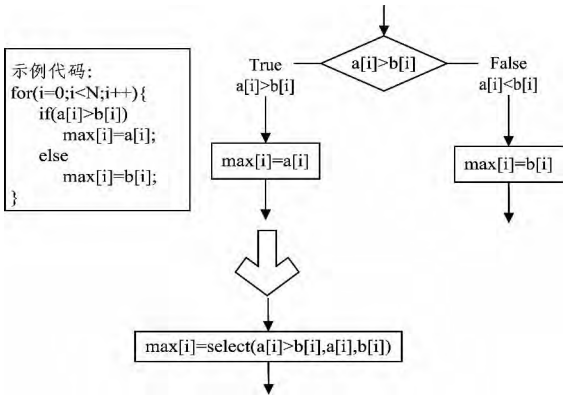

图 6 状态合并示意图

4)利用现有的回归测试集确定执行的优先顺序。Paul 等[25]于 2012 年在第 34 届 ICSE 会议上作了关于如何利用现有回归测试集提高测试效率的报告。报告中先证明了结合使用基于符号执行技术增强标准回归测试集的有效性，提供了一种关于回归测试集质量敏感度的理论和实证分析方法；并讨论了随着测试用例的增长，触发程序错误的概率的变化趋势；最后提出了一种使用轻量级符号执行机制增强现有测试集测试效应的技术。利用现有测试集可以有导向性地在潜在目标路径上执行，能较快地获得较为理想的代码覆盖率；其缺陷在于可能现有的测试集中的一组测试集仅对应一条路径，即存在大量测试集测试冗余、重复的情况，并且绝大部分测试集可能只测试到程序的一般特征，从而遗失程序在特定情形下的特征，然而这些特定情形下的特征往往是我们的分析目标。

5)基于约束的符号执行。Engler 等[26]于 2007 年提出了基于约束的符号执行方法，其目的是要抽离出单独的函数或代码区域进行分析，而不是对全程序进行分析，以提高符号执行的可用性。基于约束的符号执行将部分符号输入作为基于约束的输入，从而将目标函数隔离出来。基于约束的符号执行也不局限于对函数的抽离，对于一些影响符号执行性能的循环或者代码区域，都可以将其标记为基于约束的区域而越过。在第 24 届 USENIX Security Symposium 会议上，Da 等[27]将基于约束的符号执行引入到 KLEE 中，实现了 UC-KLEE。基于约束的符号执行方法可以极大地提高分析的针对性，通过对目标函数构造分析程序，避免陷入全程序分析的路径空间爆炸问题；其主要缺陷在于从程序中剥离出目标函数进行分析测试时，会导致对目标函数的分析不准确，从而无法分析到理想结果。

4.2 约束求解

约束求解是符号执行的基础，符号执行在程序分析上的效率很大程度上取决于约束求解的效率；而约束求解主要依赖于可满足性模理论 SMT。SMT 求解器的核心是布尔逻辑的可满足性理论，简称 SAT。理论上说，所有的 SMT 问题都可以转换成 SAT 问题，SAT 问题是历史上第一个被证明的 NP-Complete 问题。但在实际应用中，研究人员已经提出了多种约束求解的优化方法，以提高约束求解能力。当前主流的约束求解器有微软研发的 Z3 求解器[28]，其目标是解决在软件验证和软件分析过程中出现的问题。Z3 的主要应用领域包括扩展的静态检测、测试用例生成和谓词抽象。STP 求解器是面向位向量(bit-vectors)和数组处理的约束求解器，可以应对大量数组读取、深层嵌套的数组写入、线性方程及多变量的输入情况求解，在求解过程中进行迭代式的判断，提高了求解的能力[9]。SVC，CVC，CVC Lite[30]和 CVC3[31]是求解可满足性模理论问题的自动定理证明器，可被用于证明大量内置逻辑及一阶公式组合的有效性。

近年来，尽管许多研究在约束求解能力上取得了重大进展，但就目前的情况来看，约束求解的限制仍然是提高符号执行引擎可用性的主要障碍。约束求解的核心问题是将路径条件的算术约束条件转换为基本的求解器问题，其难点主要是：1)非线性整数约束往往使得路径条件不可解，带有非线性约束的约束集一般是不可判定其可解性的[32]；2)对于路径约束条件中包含的外部库函数调用的情况，求解器也无法进行处理[33]。相关研究人员也提出了针对约束求解的优化方法，其中两个突出的优化思想是：无关约束消除技术和缓存求解策略[17]。

1)无关约束消除。无关约束消除的目的是要通过分析来减少约束项的数量。一个重要的事实是，一个程序分支通常只依赖于一小部分程序变量，该分支所依赖的程序变量可能与该路径上其他约束包含的变量相互独立。因此，可以尝试从当前路径约束中识别出与当前分支结果不相关的约束并移除。

在 EXE[9]中介绍了一种约束独立性优化方法，即根据约束集中约束项所包含的约束变量的独立性，将约束集分解为独立子集，以达到简化约束集的目的。约束独立性判定在现实程序分析中有着较为广泛的应用，能有效地提高约束求解的效率。减少约束项的另一种思路是在 KLEE 中使用的隐含值具体化方法，其思想是：当一个类似 $x+1=10$ 的约束被加入到路径条件中时，后续 x 的取值在该路径上将被具体化。KLEE 将内存中的 x 置为 9，确保后续对内存中 x 值的访问都可以返回一个常量表达，以提高求解效率。无关约束消除方法能有效简化约束集，并降低约束集的维数，从而减少约束求解所耗费的时间。

2)缓存求解策略。缓存求解策略能提高约束求解性能的根据是，在进行程序分析测试过程中，经常会涉及到相似约束集的求解，尤其是与约束独立性判定方法结合使用时，缓存求解策略显得更为有效。例如：在混合测试中，求解新路径的方法是将当前执行路径中的某个分支条件进行约束取反，得到新的约束集后再进行求解，所得结果即为新路径的测试用例。在这种情况下，如果缓存了先前执行中约束公式及其解的匹配，就可以很快地借助先前的解来求解新的约束集。假设缓存中包含一个匹配：formula：$(x+y<10)\cap(x>5)\rightarrow$solution：$\{x=6,y=3\}$，而在后续的分析中遇到一个新的约束集为缓存匹配中某约束集的子集，则可以直接将缓存中该约束集对应的 solution 作为新遇到的约束集的解。另外，若后续分析中遇到的约束集是缓存中约束集的超集，如$(x+y<10)\cap(x>5)\cap(y\geqslant 0)$，则无法确定之前缓存中的结果是否可以直接使用，因此可以尝试借助先前缓存中的 solution 来判断其是否为可行解。通常，在实际分析过程中增加少量约束并不会导致先前的求解失效。对先前的求解结果进行缓存能够极

大地提高约束求解的效率，从而提高符号执行的性能。缓存求解策略是解决约束求解困难的关键方法，尤其是在混合执行中，因为相邻执行之间的约束信息仅有细微差别，因此其求解的结果理论上也只存在较小的差异。充分利用先前执行的解，能极大地缓解约束求解的压力，高效地获得可行解。

3）懒约束求解策略[34]。Ramos 等于 2015 年提出了懒约束求解策略，其核心思想是不在分析过程中对每一个遇到的分支判断都验证路径可达性，而是在该路径到达目标位置时，才通过查询求解器验证该路径的可达性并求解生成测试用例。在执行过程中，当遇到一个分支条件包含难以求解的符号操作时，需要分别对 true 分支和 false 分支继续进行搜索，同时将该约束项定义为懒求解约束项，并添加到路径条件中。当搜索过程达到目标状态时（触发错误），再进行该路径的可达性判定，如果求解结果显示其在实际执行中不可达，则舍弃该路径。懒约束求解的核心思想是推迟求解路径约束，仅对分析之后认为存在潜在错误的路径进行约束求解，以避免大量不必要的路径约束求解，使约束求解更加聚焦。其存在的缺陷在于：程序错误可能存在于程序深处，但在分析开始时就可以判定出该路径是不可达的，则后续的大量分析都毫无意义。

5　主流符号执行工具

5.1　源代码分析工具

5.1.1　EXE & KLEE

EXE[9-10] 和 KLEE[17] 是两款针对源代码的符号执行分析工具，均出自斯坦福大学的研究团队。其中，EXE 使用的是动态符号执中的执行生成测试，相较于此前的符号执行工具，其优势在于：1）可以强制执行程序的任何可达路径；2）在遇到危险操作时，求解当前路径的约束条件的结果可以是满足该约束的任意值。当一条路径终止或触发到程序错误时，EXE 通过使用 STP 求解当前路径的约束条件，自动生成测试用例。KLEE 是基于 EXE 改进开发的，运行在 LLVM 虚拟机上。其在改进符号执行的可用性方面做出了重要改进：为缓解路径爆炸问题，使用了两种启发式搜索策略，即覆盖率优化搜索（Coverage-optimized search）和随机路径选择（Random path search）。覆盖率优化搜索是指优先选择距离未覆盖指令最近的路径及优先选择新近执行到新代码的路径；与随机搜索不同，随机路径搜索是基于控制流图随机选择一条路径，并根据该路径的约束集来求解该路径的测试用例。在约束求解方面，综合使用了无关约束消除技术和缓存求解策略来提高约束求解性能。KLEE 能生成高覆盖率的测试集，在用户级程序中能达到 90％的覆盖率，而且在复杂软件中，可以通过交叉检查挖掘到更深层的程序错误。

5.1.2　DART，CUTE 和 jCUTE

DART 由贝尔实验室的 Patrice 等[43] 开发，是第一款针对 C 语言的混合测试工具。DART 的主要优势是可以对任何编译型程序进行完全自动化的测试，而不需要编写测试用例或约束代码，可以检测到诸如程序奔溃、断言违规和无终止状态等错误。CUTE 和 jCUTE 是由伊利诺伊大学开发的混合执行分析工具，其中 CUTE 基于 DART 开发，在 DART 的基础上引入了对多线程的支持；而 jCUTE 则是针对 Java 语言[35] 的。CUTE 和 jCUTE 由两个主要模块组成：1）程序插桩模块；2）用于进行符号执行、约束求解和线程调度控制的库文件。CUTE 插桩模块使用 CIL 工具，而 jCUTE 使用 SOOT 工具，对目标程序进行插桩，使用 Ip_solve 约束求解器求解路径约束，它们通过保存其自身在符号执行过程中生成的测试用例，来及在文件系统中保存的时间表，来达到程序错误可重放的目的。

5.1.3　Other tools

过去十余年里，针对源程序的符号执行工具不断涌现，比较著名的有 RubyX，CREST，PEX 和 Symbolic PathFinder 等。RubyX 是美国马里兰大学帕克分校研发的针对 Ruby 语言的符号执行分析工具[36]，它第一次将符号执行应用到了基于 Ruby-on-Rails 框架构建的 Web 应用程序分析上。CREST 由美国 UC Berkeley 大学开发，是一款针对 C 源程序的混合测试工具[18]。在 CREST 中，研究者们实现了多种启发式搜索策略，提高了混合测试的效率，后续有大量的研究基于 CREST 开展，并取得了显著成果。PEX 是微软研究院开发的针对. NET 框架下的程序的混合测试工具[37]。Symbolic PathFinder 是由 NASA 研究中心开发的面向 Java 字节码的符号执行分析工具[38]，结合使用了符号执行和模型校验两种程序分析方法。

5.2　二进制分析工具

5.2.1　angr

angr[39] 是由加利福尼亚圣塔芭芭拉大学的 Shellphish 团队研发的，用于参加 DAPRA 主办的网络空间挑战赛的二进制分析框架。angr 是用 python 开发的高度模块化的开源二进制分析框架，使用 Valgrind 的 VEX 作为中间语言。其功能包括：二进制的符号执行、智能状态合并、各类程序流图（CFG，CDG，DFG，DDG，CG 等）恢复、反汇编、值集分析、程序切片等。其主要创新点是：支持跨平台多架构；集成了程序切片和状态拟合等程序分析方法。

5.2.2　SAGE

SAGE[42] 是由微软 MSR 团队和 CSE 团队联合研发的源程序测试分析工具，结合使用了模糊测试和动态符号执行。SAGE 基于 DART[43] 开发，并在 DART 的基础上拓展了 Fuzz 的功能，使得 DART 可以测试大型应用软件，是第一个将动态符号执行应用在 x86 架构下的程序分析的系统。SAGE 提出了一种新的导向型搜索策略，称为代搜索策略（generation search）。其主要思想是：将每次符号执行生成的新的测试用例最大化，对于给定的路径约束，将该路径上的所有约束条件进行逐一取反，并将其与路径约束的前缀信息一起提交给约束求解器 Z3 进行求解，从而得到数量巨大的新测试用例，极大地提高了 SAGE 生成测试用例的效率。

5.2.3　Bitblaze & Fuzzball

Bitblaze [43] 由 UC Berkeley 大学的 Dawn 等人于 2010 年在黑客大会报告上推出，是一款二进制混合执行分析工具。Bitblaze 的 3 个核心组件是：Vine，TEMU 和 Rudder。Vine 是 Bitblaze 的静态分析组件，能提供目标二进制代码的中间表示及静态分析结果；TEMU 是 Bitblaze 的动态分析组件，负责目标二进制适配的系统仿真、二进制程序插桩和污点分析，其开发是建立在模拟处理器 QEMU 上的，目标程序及其适配的操作系统都由 QEMU 模拟仿真实现；Rudder 是 Bitblaze 的混合执行组件。FuzzBALL 是基于 Bitblaze 的静态分析组

件 Vine 开发的另一款二进制分析工具，是由 UC Berkeley 开发的面向 x86 二进制代码的在线符号执行分析工具。FuzzBALL 中提出了两种关键技术：1）利用静态分析来引导二进制程序测试用例自动化生成的技术，对待搜索的路径进行优先级判定[45]；2）利用低保真的模拟器进行高保真测试，提升路径搜索能力，搜索影响程序执行的高保真模拟器的指令空间及其状态，从而进行高覆盖率的仿真测试[46]。

结束语 2016 年美国高等研究计划署（DAPRA）主办了自动网络攻防赛（Cyber Grand Challenge），旨在建立实时自动化的网络防御系统，并且能够快速地应对大量新的攻击手法，以应对频发的网络攻击。符号执行在参赛队的自动攻防系统中起到了举足轻重的作用，被广泛应用在程序脆弱性的分析上，并涌现出了新的二进制符号执行分析框架，如圣塔芭芭拉大学 Shellphish 团队的 angr[39] 和卡内基梅隆大学 ForAllSecure 团队的 Mayhem[29]。自动网络攻防赛的举办掀起了二进制符号执行的又一波热潮。

虽然近年来各种符号执行优化技术不断被提出，符号执行的可用性和实用性也极大地提高了，但在现实软件分析应用中还面临着严峻的挑战，除了路径空间启发式搜索和约束求解之外，并行处理问题、内存建模和执行环境仿真等问题都有待进一步研究与改进。

符号执行的另一发展方向是与 Fuzzing 技术相结合，以提高程序脆弱性检测的能力。微软研究机构在 SAGE 中就已经进行了这一尝试和探索[42]，并成功将其应用在对 Office 和 Windows 等产品的错误检测上，也取得了显著的成果。2016 年，Shellphish 团队基于 angr 分析框架和 AFL 模糊测试器开发了 Driller[41]，用符号执行来增强模糊测试，为后续的研究提供了新思路。

参 考 文 献

[1] BOYER R S, ELSPAS B, LEVITT K N. Select—a formal system for testing and debugging programs by symbolic execution [C] // International Conference on Reliable Software. New York, USA, ACM, 1975: 234-245.

[2] KING J C. Symbolic execution and program testing[J]. Communications of the Acm, 1976, 19(7): 385-394.

[3] MYERS G J. Art of Software Testing[M]. New York: John Wiley & Sons, Inc., 1979.

[4] CADAR. Symbolic Execution for Software Testing in Practice-a Preliminary Assessment[C] // International Conference on Software Engineering. IEEE, 2011: 1066-1071.

[5] SEN K. Concolic testing[C] // Ieee/acm International Conference on Automated Software Engineering. ACM, 2007: 571-572.

[6] MESNARD F, ÉTIENNE P, GERMÁN V. Concolic testing in logic programming[J]. Theory & Practice of Logic Programming, 2015, 15(4/5): 711-725.

[7] PALACIOS A, VIDAL G. Concolic Execution in Functional Programming by Program Instrumentation[C] // International Symposium on Logic-Based Program Synthesis and Transformation. Springer International Publishing, 2015: 277-292.

[8] SEN K. Concolic testing: a decade later (keynote)[C] // International Workshop on Dynamic Analysis. ACM, 2015: 1-1.

[9] CADAR C, GANESH V, PAWLOWSKI P M, et al. EXE: automatically generating inputs of death[C] // Proceedings of the 13th ACM Conference on Computer and Communications Security. ACM, 2006: 322-335.

[10] CADAR C, ENGLER D. Execution Generated Test Cases: How to Make Systems Code Crash Itself[M] // Model Checking Software. DBLP, 2005: 902-902.

[11] CADAR C, SEN K. Symbolic execution for software testing: three decades later[OL]. http://www.cs.umd.edu/class/fall2017/cmsc818O/papers/symbolic-execution.pdf.

[12] CHIPOUNOV V, GEORGESCU V, ZAMFIR C, et al. Selective Symbolic Execution[C] // The Workshop on Hot Topics in System Dependability. 2009: 1286-1299.

[13] MAJUMDAR R, SEN K. Hybrid Concolic Testing[C] // International Conference on Software Engineering. IEEE, 2007: 416-426.

[14] WEN S, FENG C, MENG Q, et al. Testing Network Protocol Binary Software with Selective Symbolic Execution[C] // International Conference on Computational Intelligence and Security. IEEE, 2017: 318-322.

[15] MORAN K, LINARES-VÁSQUEZ, BERNAL-CÁRDENAS, et al. Denys Poshyvanyk[C] // 2016 IEEE International Conference on Automatically Discovering Reporting and Reproducing Android Application Crashes, Software Testing Verification and Validation (ICST). 2016: 33-44.

[16] STEPHENS N, GROSEN J, SALLS C, et al. Driller: Augmenting fuzzing through selective symbolic execution[C] // 23nd Annual Network and Distributed System Security Symposium (NDSS 2016). 2016.

[17] CADAR C, DUNBAR D, ENGLER D R. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs[C] // Usenix Conference on Operating Systems Design and Implementation. USENIX Association, 2008: 209-224.

[18] BURNIM J, SEN K. Heuristics for Scalable Dynamic Test Generation[C] // IEEE/ACM International Conference on Automated Software Engineering. 2008: 443-446.

[19] SEN K, AGHA G. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools[C] // International Conference on Computer Aided Verification. 2006: 419-423.

[20] BOONSTOPPEL P, CADAR C, ENGLER D. RWset: Attacking Path Explosion in Constraint-Based Test Generation[C] // Theory and Practice of Software, International Conference on TOOLS and Algorithms for the Construction and Analysis of Systems. Springer-Verlag, 2008: 351-366.

[21] GODEFROID P. Compositional dynamic test generation[C] // Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2007). New York, USA, ACM, 2007: 47-54.

[22] HANSEN T, SCHACHTE P, SØNDERGAARD H. Runtime verification. chapter State Joining and Splitting for the Symbolic Execution of Binaries[M] // Berlin: Springer-Verlag, 2009: 76-92.

[23] KUZNETSOV V, KINDER J, BUCUR S, et al. Efficient state merging in symbolic execution. In Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2012). New York, USA, ACM, 2012: 193-204.

[24] AVGERINOS T，REBERT A，SANG K C，et al. Enhancing symbolic execution with veritesting[C]// International Conference on Software Engineering. ACM，2014：1083-1094.

[25] MARINESCU P D，CADAR C. make test-zesti：a symbolic execution solution for improving regression testing[C]// International Conference on Software Engineering. IEEE，2012：716-726.

[26] ENGLER D，DUNBAR D. Under-constrained execution：making automatic code destruction easy and scalable[C]// International Symposium of Software Testing and Analysis. ACM，2007：1-4.

[27] RAMOS D A，ENGLER D. Under-constrained symbolic execution：correctness checking for real code[C]// Usenix Conference on Security Symposium. USENIX Association，2015：49-64.

[28] MOURA L D，BJØRNER N. Z3：An Efficient SMT Solver[M]// Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg，2008：337-340.

[29] SANG K C，AVGERINOS T，REBERT A，et al. Unleashing Mayhem on Binary Code[C]// Security and Privacy. IEEE，2012：380-394.

[30] BARRETT C，BEREZIN S. CVC Lite：A New Implementation of the Cooperating Validity Checker[C]// Computer Aided Verification，International Conference（CAV 2004）. Boston，MA，USA，DBLP，2004：515-518.

[31] BARRETT C，TINELLI C. 19th International Conference on Computer Aided Verification（CAV'07）[M]// Damm W，Hermanns H，eds. Berlin：Springer，2007：298-302.

[32] ENDERTON H B，COMPUTABILITY M D，UNSOLVABILITY. Hilbert's Tenth Problem is Unsolvable[M]// American Mathematical Monthly. 1973：233-269.

[33] XIAO X，XIE T，TILLMANN N，et al. Precise identification of problems for structural test generation[C]// ICSE 2011. ACM，2011：611-620.

[34] RAMOS D A，ENGLER D. Under-constrained symbolic execution：correctness checking for real code[C]// Usenix Conference on Security Symposium. USENIX Association，2015：49-64.

[35] SEN K，AGHA G. CUTE and jCUTE：Concolic Unit Testing and Explicit Path Model-Checking Tools[C]// International Conference on Computer Aided Verification. Springer-Verlag，2006：419-423.

[36] CHAUDHURI A，FOSTER J S. Symbolic security analysis of ruby-on-rails web applications[C]// ACM Conference on Computer and Communications Security. ACM，2010：585-594.

[37] TILLMANN N，DE HALLEUX J. Pex：white box test generation for .NET[C]// Tests & Proofs，Second International Conference. Tap，Prato，Italy，2008：134-153.

[38] PĂSĂREANU C S，RUNGTA N. Symbolic PathFinder：symbolic execution of Java bytecode[C]// IEEE/ACM International Conference on Automated Software Engineering（ASE 2010）. Antwerp，Belgium，DBLP，2010：179-180.

[39] SHOSHITAISHVILI Y，WANG R，SALLS C，et al. SOK：（State of）The Art of War：Offensive Techniques in Binary Analysis[C]// 2016 IEEE Symposium on Security and Privacy（SP）. IEEE，2016：138-157.

[40] YAN S，WANG R，HAUSER C，et al. Firmalice-Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware[C]// Network and Distributed System Security Symposium. 2015.

[41] STEPHENS N，GROSEN J，SALLS C，et al. Driller：Augmenting Fuzzing Through Selective Symbolic Execution[C]// Network and Distributed System Security Symposium. 2016.

[42] LEVIN M Y，LEVIN M Y，MOLNAR D. SAGE：whitebox fuzzing for security testing[OL]. https://www.researchgate.net/publication/220309880_SAGE_Whitebox_Fuzzing_for_Security_Testing.

[43] GODEFROID P，KLARLUND N，SEN K. DART：directed automated random testing[C]// ACM Sigplan Conference on Programming Language Design and Implementation. ACM，2005：213-223.

[44] SONG D，BRUMLEY D，YIN H，et al. BitBlaze：A New Approach to Computer Security via Binary Analysis[C]// Information Systems Security，International Conference（ICISS 2008）. Hyderabad，India，DBLP，2008：1-25.

[45] BABIĆ D，MARTIGNONI L，MCCAMANT S，et al. Statically-directed dynamic automated test generation[C]// Proceedings of the 2011 International Symposium on Software Testing and Analysis. ACM，2011：12-22.

[46] MARTIGNONI L，MCCAMANT S，POOSANKAM P，et al. Path-exploration lifting：Hi-fi tests for lo-fi emulators[C]// ACM SIGARCH Computer Architecture News. ACM，2012：337-348.

（上接第 10 页）

[60] SHADBOLT N，BERNERS-LEE T，HALL W. The semantic web revisited[J]. IEEE Intelligent Systems，2006，21(3)：96-101.

[61] BERNERS-LEE T，CHEN Y，CHILTON L，et al. Tabulator：Exploring and Analyzing linked data on the Semantic Web[C]// Proceedings of the 3rd International Semantic Web User Interaction Workshop. 2006.

[62] HANNEMANN J，KETT J. Linked Data and Libraries[OL]. [2011-01-18]. http://www.ifla.org/ files/hq/papers/ ina76/149-hannemann-en.pdf

[63] MALMSTEN M，李雯静. 将图书馆目录纳入语义万维网[J]. 数据分析与知识发现，2009，3(3)：3-7.

[64] SUMMERS，ANTOINE，ISAAC，等. LCSH，SKOS 和关联数据[J]. 数据分析与知识发现，2009，3(3)：8-14.

[65] SCHMACHTENBERG M，BIZER C. Linking Open Data cloud diagram[OL]. http://lod-cloud.net/.

[66] WANG C，MARSHALL A，ZHANG D，et al. ANAP：an integrated knowledge base for Arabidopsis protein interaction network analysis[J]. Plant physiology，2012，158(4)：1523-1533.

[67] BRANDÃO M M，DANTAS L L，SILVA FILHO M C. AtPIN：Arabidopsis thaliana protein interaction network.[J]. Bmc Bioinformatics，2009，10(1)：1-7.

[68] SAIER JR M H，REDDY V S，TAMANG D G，et al. The transporter classification database[J]. Nucleic acids research，2013，42(1)：251-258.

[69] DAI X，ZHAO P X. psRNATarget：a plant small RNA target analysis server[J]. Nucleic Acids Research，2011(39)：155-159.

[70] BARRETT T，WILHITE S E，LEDOUX P，et al. NCBI GEO：archive for functional genomics data sets-update[J]. Nucleic acids research，2012，41(1)：991-995.