Towards Optimal Concolic Testing (ICSE 2018)

<https://www.icse2018.org/details/icse-2018-Technical-Papers/4/Towards-Optimal-Concolic-Testing>

# 摘要

具体-符号测试（Concolic Testing）整合了具体执行（例如随机测试）和符号执行用于产生测试用例。它有时会表现出相较于随机测试或符号执行更高的成本效益。一个具体-符号测试策略是一个方程，它决定了什么时候使用随机测试或符号执行，而且如果是后者的话，还要决定符号执行在哪个路径。许多基于启发式的策略被提出，而什么是最佳的具体-符号测试策略仍是一个未解决的问题。本工作为解决这个问题做出了两点贡献。第一，我们展现最优策略能根据程序路径的概率和约束求解的成本被定义。定义最优策略问题进一步被简化到一个有成本的马尔科夫决策过程的模型检验问题。第二，鉴于确定最优策略的复杂性，我们设计了一个贪婪算法来逼近最优策略。我们开展了两组实验。一组是基于随机生成的模型，另一组是给予一套C程序。结果表明现存的启发式方法有很大提升空间，而且我们的贪婪算法常常由于这些现有的启发式方法。

# 1. 介绍

具体-符号测试，也称作动态符号执行，是具体执行（也称作测试）和符号执行[22, 41]的集成。具体执行和符号执行互补。 一方面，具体执行在计算代价低。 也就是说，我们根据所有测试输入的先验概率分布保持采样测试输入，并且具体地用测试输入执行程序直到满足某些测试覆盖标准。问题在于，如果某个程序路径的概率非常低，则必须对大量测试输入进行采样以覆盖程序路径。另一方面，符号执行通过识别必须满足的约束来解决该问题，以覆盖程序路径并解决约束来获得测试输入。换言之，用符号执行覆盖程序路径的概率是1。问题是符号执行通常计算代价高。直观地，有效的具体-符号测试策略应该是利用符号执行执行那些低概率程序路径，并具体执行那些路径条件难以求解的程序路径。

什么是最佳的具体-符号测试策略仍是一个未解决的问题。文献中，已经很多尝试来解决这个问题[6,7,23,33,38,42]。例如，几种启发式方法得到提出来回答问题：哪个程序路径（所有程序路径中）我们要在具体-符号测试中符号执行？举几个例子，Burnim et al. 提出CFG 策略[6]，计算从一个执行路径中的分支到任何未覆盖的条件并选择一个有最小距离的分支。Godefroid et al.[23] 提出世代策略，测量执行路径中每个分支的增量覆盖的增益，并通过扩展具有最高覆盖增益的分支来引导搜索。Li et al.[33]引入一种技术，在较少经过的路径上使用符号执行。虽然现有的启发式方法在实验上有效，但更加高的性能是否能够达到或距最佳性能有多远仍不清楚。

此外，现有工作在很大程度上忽略了问题的其他关键部分，即我们如何在具体执行和符号执行之间切换以实现最佳性能？据我们所知，这个问题只在近期被讨论[3,4,45]。作者基于程序概率角度，比较了随机测试和系统测试方法（包括但不限于符号执行），并呈现出一种混合策略，从随机测试切换至系统测试当后者被认为能在单位时间发现更多错误。然而，他们的方法对系统测试方法采取了非常抽象的观点，而且也不考虑如应用符号执行的不同策略。此外，他们的算法非常高层次，只在模拟的模型上被验证。

在这项工作中，我们的目标是开发一个框架，让我们能够定义和计算最佳的分析测试策略。即，我们的目标是系统性地回答何时应用具体执行，何时应用符号执行，以及将符号执行应用于哪个程序路径。特别地，我们做出以下技术性贡献。第一，我们展现最优具体-符号测试可以基于程序行为的概率抽象来定义。第二，我们展现识别最优策略的问题可以简化为具有成本的马尔可夫决策过程的模型检验问题。就结果而言，我们可以重用现有的工具和算法来解决问题。第三，我们用一组模拟实验来验证现有的启发式算法，并展现它们有很大的改进空间。第四，鉴于计算最优策略的高度复杂度，作者提出了一种近似最优策略的贪婪算法。我们根据模拟实验和C程序实验对贪婪算法进行了实证评估，并证明它比KLEE[7]中现有的启发式算法获得更好的性能。

论文的剩余部分组织如下。第2节定义了研究问题，并用一个例子说明了它的相关性。第3节化简问题为模型检验问题，并将现有启发式方法和最优策略进行对比。第4节设计了一个贪婪算法来逼近最优策略。第5节提出我们的实现，并验证这个贪婪算法。第6节回顾相关工作。第7节下结论。

# 2. 问题定义

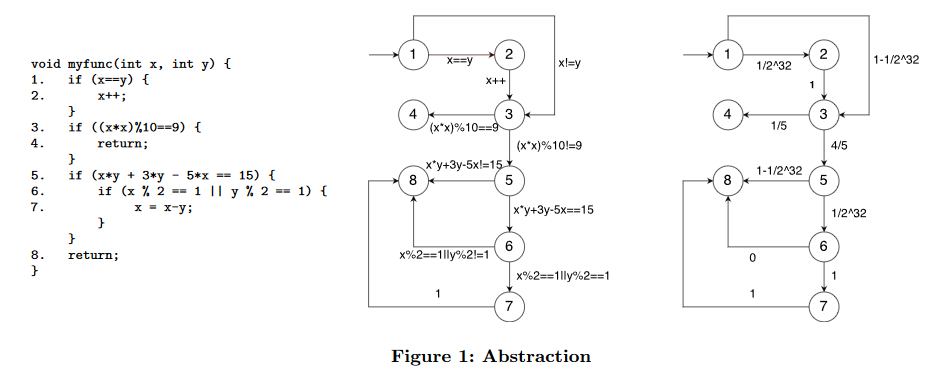
下面我们定义问题，不失一般性，我们定义如下的程序（如JAVA/C）。

定义2.1. 一个程序是一个标号迁移系统。

* 是一个控制位置的有限集合。
* 是一个独特的入口点（也就是程序的开始）。
* 是一个变量有限集合。
* 是的一个捕获一组初始估值的谓词。
* 是一个迁移方程，其中每一个迁移用一个形式为的卫式命令标注，是一个卫式条件，是一个更新变量估值的方程。

一个的具体执行（也称作一个测试）是一个序列*π* = ⟨(*v*0*,c*0)*,gc*0*,*(*v*1*,c*1)*,gc*1*,*··· *,*(*vk,ck*)*,gck,*···⟩，其中*vi* 是的估值，*ci* ∈ *C*, *gci* = [*gi*]*fi*是一个卫式命令，，还有对于所有有*vi*+1 = *fi*(*vi*)，，。我们说*π*覆盖一个控制位置c当且仅当c在这个序列中。一个控制位置c是可到达的当且仅当存在一个覆盖c的具体执行。初始变量估值也被称为测试用例。

的一个（根）程序路径是一个连接的迁移序列，*π* = ⟨(*c*1*,gc*1*,c*2)*,*(*c*2*,gc*2*,c*3)*,*··· *,*(*ck,gck,ck*+1)⟩，*c*1 = *init* ，对于所有有(*ci,gci,ci*+1) ∈ *T*，相应的路径条件*PC*(*π*) = ∃*v*2*,*··· *,vk*+1*. g*1∧ (*v*2 = *f*1(*v*1))∧*g*2∧···∧*gk*∧(*vk*+1 = *fk*(*vk*)).我们用*path*(𝒫)标记程序P的所有路径。



例2.2. 图1展现一个简单的Java程序。对应的迁移系统在图中间，为了可读性略去命令。迁移系统包含8个控制位置，对应程序中的8个编号行。为了简单起见，我们假设每条线都是原子线。初始条件是其中是所有整数的集合。

为了简单起见，我们假定目标是生成测试用例，这样相应的具体执行覆盖所有可到达的控制位置（即100％语句覆盖率）。文献中，有许多测试用例生成方法[11,12,26]。本文中，我们专注于两种方法生成测试用例。

一个是随机测试。为了开展随机测试，我们修复所有测试用例中的一个先验分布μ，并每次根据μ随机采样测试用例。随后，使用采样测试用例执行程序直到完成执行。例如，我们假定图1中所有测试用例服从均匀分布，随机测试是随机生成x和y，之后具体执行这个程序。就时间而言，随机测试的成本通常很小。在这项工作中，我们简单地假设成本是1个时间单位。假设每个测试用例都与μ中的某非零概率相关联，最终显示我们可以通过随机测试枚举所有测试用例并覆盖所有可到达的控制位置是微不足道的。不幸地，实践中我们的时间和预算有限，因此我们可能无法用有限数量的随机测试案例覆盖某些控制位置。例如，对于x和y的所有可能值中的均匀概率分布，平均需要个随机测试用例来覆盖图1中的第2行。

另一生成测试用例的方法是符号执行[12]。给定一个程序路径，使用约束求解器用于检查路径条件的可满足性，并且如果它是可满足的就构造测试用例。之后，我们使用测试用例执行程序直到它完成执行。符号执行有时比随机测试成本效率高。例如，用约束求解器Z3[14]，我们可以很容易地解决的路径条件（即x==y）来访问图1中第2行来产生需要的测试用例。但是，符号执行可能并不总是具有成本效益。例如，为了获得覆盖第4行的测试，我们可以应用符号执行来解决包括在第3行条件的条件。由于非线性约束，它可能是非平凡的。相比之下，生成随机测试用例以满足第3行的条件要容易得多，即平均需要5个随机测试用例。通常，符号执行的成本远大于随机测试的成本，因为约束求解可能是很耗时的。

此外，当用符号执行来生成测试用例以覆盖某个控制位置时，我们可以求解以控制位置路径结束处的路径条件或其前缀的路径条件。例如，为了覆盖第7行，我们可以求解由第1,3,5,6和7行组成的路径，或者由第1,3,5和6行（一次或多次）组成的路径来生成测试用例。后者可能更具成本效益，因为要解决的约束的条款较少。在这个特定的例子中，解决后者一次就足以覆盖第7行。

具体-符号测试，是具体执行（也称作测试）和符号执行的集成。在本工作中，我们定义了一种策略让具体-符号测试成为一种方程来在随机测试或符号执行（在某个路径上）之间重复地生成一个选择，直到达成测试目标。有两个极端：（1）始终应用随机测试，和（2）对每个程序路径应用符号执行。这里有许多替代品[6,7,23,33,38,42]。现有的具体-符号测试引擎已采用多种策略（例如KLEE[7]，Pex[47]和JDart[34]）。如上我们展示的，对于某些程序，一种策略可能比其他策略更具成本效益。例如，对于图1所示的示例，一个“更好”策略将符号执行应用于由第1行和第2行组成的路径（以覆盖第2行），将符号执行应用于由1,3,5和 6（覆盖第7行），并应用随机测试来覆盖其余的路径。之后这个问题是如何比较不同的策略。在本工作，我们调查了不同的具体-符号测试策略的有效性，并回答了以下未解之谜。

RQ1: 对于给定的一个程序，具体-符号测试的最优策略是什么？

RQ2: 我们能否有效率地计算出最优策略吗？

RQ3: 现有策略是否逼近最佳策略？

RQ4: 是否有可能设计出一种实用的算法来逼近最优策略？

RQ5: 如果RQ4的答案是肯定的，那么该算法与现有的启发式算法相比怎么样？

我们在后续章节回答这些问题。

我们注意到，在这项工作中我们不考虑使用具体值简化复杂的符号约束的策略。此外，我们假定路径条件编码和解决是完美的，因此是不发散的。考虑这些会使讨论更加复杂化因此我们把它留给今后的工作。

# 3. 最优策略

本节中，我们展示最佳的具体-符号测试策略能够基于程序路径概率和解决约束的代价定义。此外，它能通过模型检验计算出来。

## 3.1 马尔科夫抽象

为了回答RQ1，我们首先以马尔科夫链的形式开发一个程序的抽象。

定义3.1. 一个（标记的）离散时间马尔可夫链（DTMC）是一个元组其中S是一个状态的有限集合，一个标记的迁移概率方程，对于所有有，是初始概率分布，有。

如果没有从状态的外向迁移，我们就管s叫沉没状态。我们常用表示当前状态s访问s’的条件概率。条件概率也被称作一步迁移概率。一条路径M是一个状态序列，我们用标记的集合。让标记所有的路径M，。表示的概率，使所有一步迁移概率的乘积。给定一个有限路径，我们用表示序列终态，表示倒数第二态。如果是沉没状态的话有限路径是极大。是所有M的最大路径，是所有M从s开始的最大路径。此外，我们说如果中的每个状态至多只出现一次是不重复的。记作所有以状态s为终结的有限路径。所有中的路径的累积概率是到达s的概率，为了简便记作，代表所有s开始s’结束的有限路径。表示所有中的路径的累积概率。

在下文中，本文开发了DTMC对程序的解释，这形成了后续讨论的基础。

定义3.2. 令是一个程序，是所有测试输入的一个先验概率分布。DTMC对的解释是一个DTMC，S中的一个状态是其中v是V的估值以及l是C中的一个控制位置。定义如下：当且仅当存在一个卫式命令*gc*= [*g*]*f*有，，否则。

注意如上的定义中，除了初始概率分布μ之外，每个一步迁移具有概率1或0。我们的最优具体-符号测试策略是基的一个特定抽象来定义的，即抽象出变量估值的抽象，定义如下。

定义3.3. 令是一个程序，是它的DTMC解释。的抽象DTMC解释是一个DTMC，。这是有用的因为本工作中我们专注于条件覆盖。

* 当是；其他情况等于0
* 定义如下：对所有,是：

直观地，是访问的概率并紧接着，超过达到的概率。例如，图1右侧所示的DTMC是左侧程序的数据抽象DTMC解释，其中程序中的每个控制位置成为DTMC中的状态，并且两个控制位置之间的每个控制流与相应的条件概率。 例如，概率被标记为从状态1到2的迁移，表明在状态1之后访问状态2的概率是（如果我们假设所有测试输入之间的均匀分布）。

以下命题表明到达控制位置l的概率保留在中。

命题3.4. 令是一个程序，是所有测试输入的一个先验概率分布。对于所有，其中是V所有可能估值的集合。

命题的正确性能通过展示到达任意的概率被建立：

的测试执行能自然地映射到的一条路径。例如，向图1给定程序输入x=y=0的测试执行被映射到由状态1，2，3，5，8组成的路径。我们称测试执行覆盖的一个状态当它覆盖的相应控制位置。此外，的一个路径独特地相应的一个程序路径。

## 3.2 最优策略

回忆记得具体-符号测试策略是是不同测试用例生成方法之间的一系列选择。在这项工作中，我们定义了用于选择的空间：

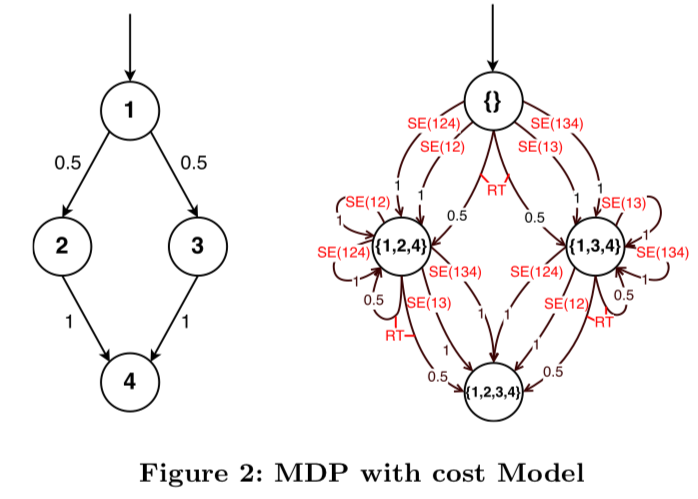
表示随机测试，表示通过求解和路径p联系的路径条件的符号执行。为了比较不同选择的成本，我们需要一种衡量它们的方法。本工作我们专注于时间开销。令为一个函数，输入返回他的时间开销。为简单起见，生成随机测试用例的时间开销设置为1个单位。的时间开销包括编码/求解路径条件的时间成本。

我们就覆盖中的状态集合的概率而言衡量一个选择的有效性。给定一个选择和的状态集合，我们能用如下随机测试计算出覆盖特定状态集合的概率。

如图1所示例子，是，是0因为同时覆盖1，4，5的测试用例。

如果选择符号执行程序路径，即，我们知道所有路径中的状态必被覆盖。让是所有从路径p的最后一个状态开始，包括p在内的覆盖所有和仅在X中的状态当最大路径集合。覆盖所有和仅在有的X中的状态的概率被记作，定义如下

如图1所示例子，是，也就是通过符号执行路径1和3，我们得到覆盖状态1，3，5的概率是。另一个例子是0因为我们必须覆盖3。

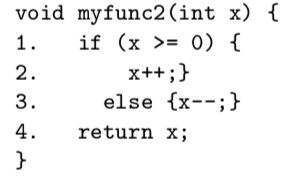


在这项工作中，我们假设可以根据某些状态是否已被覆盖来做出选择。直观上这说得通，因为如果路径上的所有状态都已被覆盖，那么最好不要将符号执行应用于该路径。策略因此是一种函数，它将关于中的每个控制位置是否已被覆盖的输入信息，并返回测试用例生成方法的选择。为了系统地比较不同的具体-符号测试策略，我们以具有代价的马尔可夫决策过程（MDP）的形式构建以下模型。

定义3.5. 令是程序的数据抽象DTMC解释。我们定义是一个有代价的MDP，有

* ，其中是S的幂集，即中的每一个成员是S中的一个状态集合（即那些已被覆盖的）。
* 是初始状态，是
* 被定义有其中和是概率分布对于所有有。被上面(1),(2)定义。
* 对于每一个，由方程定义的与一个代价联系。

例如，给定下面的简单程序。



我们能获得如图2左侧所示的数据抽象马尔可夫模型，图2右侧展示对应的。初始状态是∅，即没有覆盖任何状态。在初始状态下应用，我们可以得到一个概率分布有概率为0.5达到状态{1,2,4}（即覆盖状态1,2和4），概率为0.5达到状态{1,3,4}。如果不是这样，在路径⟨1,2⟩上进行符号执行（即SE（12）），我们有达到状态{1,2,4}的概率为1。 注意如果我们在状态{1,2,4}的路径⟨1,2⟩上应用符号执行，我们再次以概率1到达{1,2,4}，这由状态{的自循环过渡表示{1,2,4}。假设， 而，我们可以计算出基于每种选择的累积代价的预期代价。 例如，总是应用RT的预期代价是2，而应用然后的预期代价是4。

根据定义3.5，我们能看出具体-符号测试的策略和的原则是等价的，也就是，一个从到的方程，下面回答RQ1：

|  |
| --- |
| RQ1的答案：最优策略是具有最小期望代价的的原则。 |

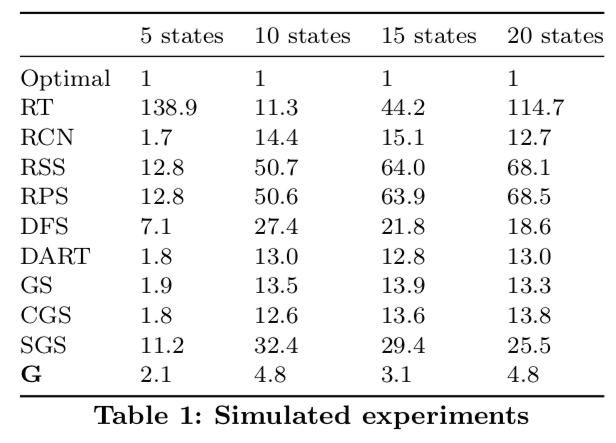
例如，在图2所示的示例中，最优策略是一直应用RT的策略（具有预期代价2）。 因此，找到最优策略的问题被解决为找到具有最小预期代价的策略的问题，这可以使用现有的方法[27]，例如值迭代，策略迭代或解决线性编程问题来解决。因此，找到最优策略的计算复杂性受到识别最优原则的复杂性的限制。

|  |
| --- |
| RQ2的答案：识别最优策略的复杂度是与中状态数量呈强多项式相关的，是与中的控制位置的数量呈指数相关的。 |

## 3.3 验证现存的启发式方法

下文中，我们展开实验来回答RQ3。也就是，我们用最佳策略的性能对比启发式方法[6, 7, 23, 33, 38, 42]的性能。目标是看看是否现存的启发式方法是原则上有效的。

我们随机的生成马尔可夫链模型集合（不存在不可达到状态）作为程序的抽象。由于计算最佳策略的高度复杂度，我们仅使用方法[46]生成包括5到20个状态的模型。对于0.5概率下每一个状态，我们生成一个分支，即期望分支密度是0.5。我们为每一个迁移随机生成一个迁移概率。为模仿低概率程序路径，对于5状态模型，我们生成概率低至1e-4的迁移（为避免没有低概率迁移），概率为0.8，0.2的概率（为避免没有过多低概率迁移）生成10，15，20状态模型。为简化实验，我们把每一个模型中的迁移和1000以内的正整数代价联系起来，而非把符号执行代价和每一个路径联系起来。我们为每个马尔可夫链建立相应的有代价MDP模型，用PRISM[32]计算最佳策略。



结果如表1所示，第一列是策略，剩下的是50个5状态马尔可夫链、50个10状态马尔可夫链，等等。行optimal是最优策略的期望代价，标准化为1.剩下的行是随机测试的结果（RT），KLEE中的四个策略：默认随机覆盖新的(RCN), 随机状态搜索(RSS), 随机路径选择(RPS), 深度优先搜索(DFS)，DART中的直接的自动随机测试，SAGE中的世代搜索（GS），上下文引导搜索（CGS）[42]，和子路径引导搜索（SGS）[33]。SGS中子路径长度被设置为模型中总路径数的20%。最后一行现在忽略掉。

我们用Java实现所有方法。对于每个马尔可夫链模型，我们重复每个策略1000次并获得平均代价（以覆盖所有状态）。注意对于随机测试，覆盖所有状态要花费极长的时间，因此我们设置1000000个测试用例的限制。从结果我们观察到所有现存的启发式方法有相较于最优开销的巨大的开销。最优表现的启发式方法的开销也比最优开销高出一阶。在所有策略中，每次都随机测试的策略在20态时表现最糟。结果表明现存的启发式方法都有很大提升空间。注意表1的结果应加盐处理因为他们给予随机生成的马尔可夫链模型，它们不能代表真的程序。

|  |
| --- |
| RQ3的答案：现存的启发式方法都能被提升。 |

# 4. 逼近最优

基于第3节中的讨论，很明显实际中确定最优策略是不可行的，因为其高度复杂性以及识别程序路径概率和符号执行成本困难。在下文中，我们提出了一种在实践中逼近最优策略的方法。我们的提议包括一种近似的方法，一种近似方程的方法，以及一种用于识别最优策略的贪婪算法。

## 4.1 估计和方程

下面，我们呈现一种估计的方法。注意，这是最近一系列被称为概率符号执行的研究的主题[15, 18, 21]。然而，概率符号执行有着很高的复杂度（由于隐含着模型计数技术[9]）。

因此，我们应用轻量级方法，也就是，我们估计基于已经获得的测试案例。我们想要解决的根本问题是：如果我们观察到某些事件（即覆盖某些程序路径的测试用例），我们如何估计所见事件的概率和那些看不见的事件（即覆盖其他程序路径的测试用例）的概率？这个问题已经被研究数十年，有大量的方法得到提出，例如拉普拉斯估计[13]，Good-Turing估计[19]。我们请读者查阅[19]以全面讨论什么时候不同的估计是有效的。下面，我们展示怎么基于拉普拉斯估计来估计。

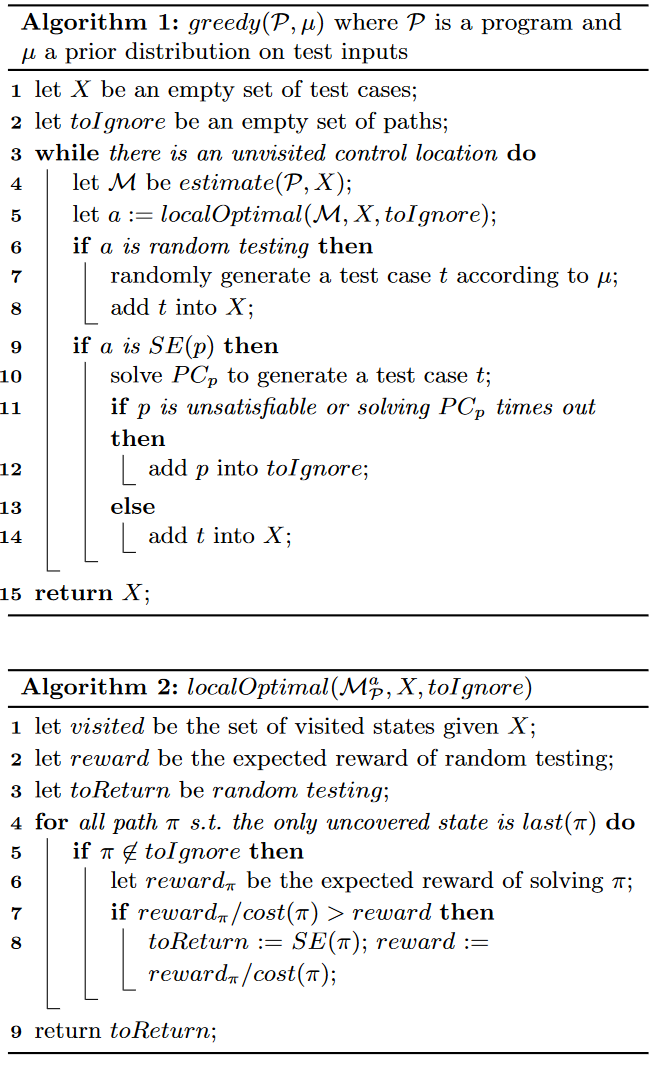
假设我们已经获得了一组测试执行，我们可以如下估计。

定义4.1. 给定任何状态，让为X中状态s被样本访问的次数。对于任何，让为X中从s到t一部迁移的次数。对于任何状态s，如果s到达中的另一个控制位置t是不可能的，我们设置为0；否则，拉普拉斯估计设置为，其中n是s可以一步达到的状态的总数。

直观上，如果一个从s到t的迁移（即一个控制流）因为太小而没有在X中被观察到，拉普拉斯估计设置迁移概率为。容易看到无线数量的样本下，估计的收敛到实际的。下面，我们用*estimate*(𝒫*,X*)表示估计的。

估计的方程，也就是解决约束的代价，是高度不平凡的，因为如Z3[14]这样的约束求解器采用了复杂的约束求解技术。这本身就是一个研究课题[29,31]。本文中，我们采用[29]中的方法，工作方式如下。第一，[29]的作者收集了通过分析一组符号执行的实际程序产生的求解约束的时间成本。假设求解约束的代价是约束中原始运算（如加法、乘法）的加权和，之后他们通过函数拟合估计每一种原始运算的权重。随后，给定一个约束c，它的求解代价被估计为所有c中原始运算的加权和。例如，如果c是a\*b>0，他的求解代价是乘法和大于比较的加权代价之和。详见[29]。

## 4.2 贪婪算法



即使有了的合理估计和方程，识别最优策略的算法仍过于复杂（见RQ2的解答）。下面，我们呈现一个复杂度更低的贪婪算法。想法是在运行中估计并应用测试用例生成方法，方法基于估计成本效益最高的方式改进测试覆盖。

细节如算法1。在第一行，我们从一个测试用例空集开始。在第二行，我们初始化集合toIgnore来存储符号执行忽略的路径。3到14行的循环迭代生成测试用例直至达到覆盖标准。每次迭代中，我们先在第四行构建一个的估计。随后，我们调用函数localOptimal选择局部最优测试生成方法。如果选择的是随机测试，我们在第七行产生一个随机测试用例；否则，我们使用符号执行来选择程序路径。如果选择的路径是不可行的或求解路径条件超时，我们就把这条路径添加到toIgnore。

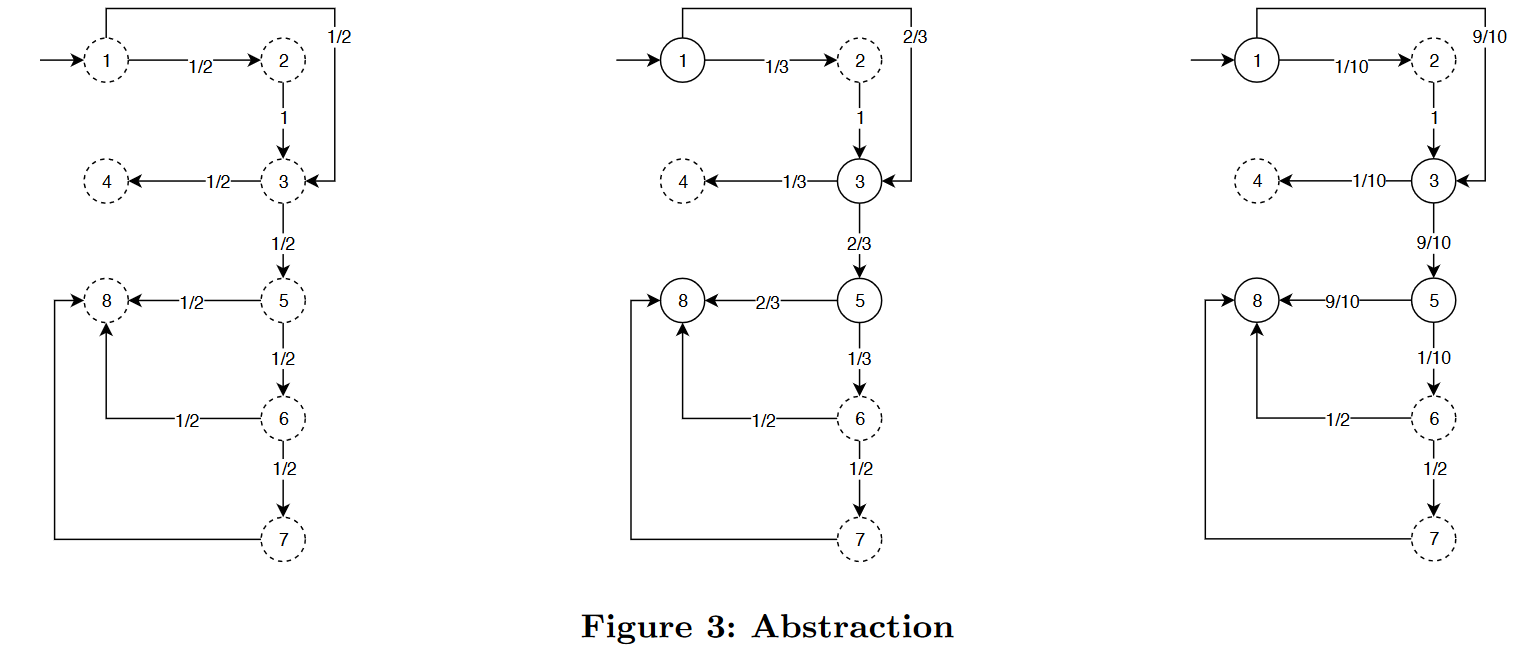
*localOptimal*(ℳ*,X,toIgnore*)如算法2，直观地，我们将测试生成方法的“奖励”定义为预期将由新生成的测试用例覆盖的未覆盖状态的数量，并选择具有每单位代价的最大期望奖励的方法。在第二行，我们基于当前估计ℳ = (*S,Pr,µ*)先计算随机测试的期望奖励。它由带奖励扩展的M（即1单元奖励与1个未被访问状态联系起来）和使用现存方法[2]求解期望问题的奖励。下面，我们展示是怎么通过求解一个方程组来解决的。

让为访问s的奖励。我们建立如下方程组：

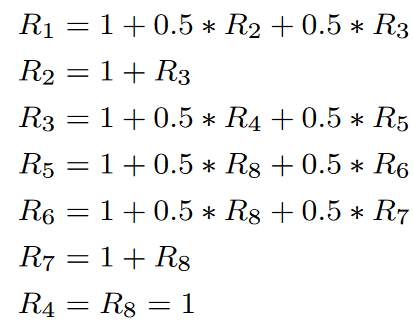
随机测试的期望奖励是Σ*s*∈*S*{*µ*(*s*)× *Rs*}，注意因为我们的目标是覆盖每一个状态，所以我们将访问每个未被访问状态的奖励关联起来。

接着，我们将随机测试的期望奖励与符号执行的期望奖励进行比较。理想情况下，我们会计算符号执行每条路径的代价以及相应的奖励，然后选择收益最高的。然而，这种路径的数量通常很大（也就是如果存在循环则无限大）。因此，我们启发式地关注除了结束状态之外不包含未覆盖状态的路径。这样，如果应用符号执行，则保证访问至少1个未被访问状态。注意这与[12,22]相似，我们假定对于每个循环的迭代次数限制已给定，且我们只考虑迭代次数更少的路径。将符号执行应用于以状态s结束的路径的期望奖励表示为，其可以使用上面讨论的相同的方程系统来获得。

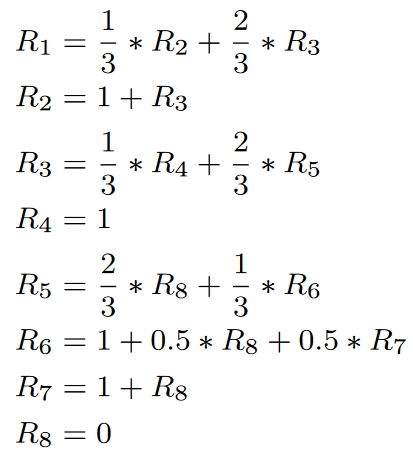
细节如算法2（4到8行）。在第五行，我们检查是否被选择的路径π被忽视。如果没有，我们通过求解相同的方程组来获得，其中s是π的终态，来计算求解π的期望奖励。直观上，这是因为通过求解路径π，我们有可能访问s并获得其所有期望奖励。也就是，如果last(π)是s，*rewardπ* 就是。在第七行，我们对比每单位代价的奖励（方程的估计在节4.1讨论过）和当前最优选择，并保持最好的那个。注意我们假设路径条件是精确的。如果通过求解路径条件产生的测试输入发散并因此没有达到s，我们把这条路径也加入toIgnore。



下面，我们将阐明该算法如何适用于图1所示的程序。为了阐明目的，我们假设求解线性（不）相等或它们的连接具有时间代价4，求解非线性等式的代价为10; 并且求解非线性等式的布尔组合的代价为50。最初，由于我们还没有测试执行，估计M显示在图3的左侧，其中未覆盖的状态是短划线的。注意，来自相同状态的所有向外迁移具有相同的概率。基于这种估计，我们通过求解以下方程系统来计算随机测试的期望奖励。



随机测试的期望奖励是 ，是4.875。因所有状态都是未被访问的，我们为符号执行选择的候选路径是仅包含状态1的路径。由于路径条件为真，因此对此路径应用符号执行与随机测试相同。注意，这意味着我们总是从随机测试开始。假设我们生成的随机测试用例覆盖控制位置1,3,5和8。然后更新估计，如图3中间所示。接下来，我们通过求解以下方程组来计算随机测试的期望奖励。



我们有R1=1，R2=5/3，R4=1和R6=1.5。符号执行的候选路径包括从1到2的路径，从1,3到4的路径，以及从1,3,5到6的路径。成本分别为4,50和50。期望奖励分别为5/3,1和1.5。因此，所选方法是随机测试。为简单起见，假设前8个随机测试执行全部覆盖1,3,5和8。就结果而言，M的更新如图3右侧所示。随机测试的期望奖励计算为0.335，然而解决从1到2的路径的预期回报是1.235。我们因此得出结论，后者更具成本效益（每代价奖励0.30875），因此将符号执行应用于路径。直观地，我们只有在覆盖路径需要大量随机测试用例时才会从随机测试切换到解决某些程序路径，这将产生比符号执行更大代价。

算法1的复杂度是合理的。为了选择正确的测试用例生成方法，每一轮中，我们付出解决方程系统的代价，方程组的变量数等于程序中控制位置的数量。现代方程组求解器通常相当有效并且开销是合理的。 我们可以进一步优化求解方程组的算法，因为只有当s能够达到新覆盖的状态时，才会在一次迭代后发生变化。

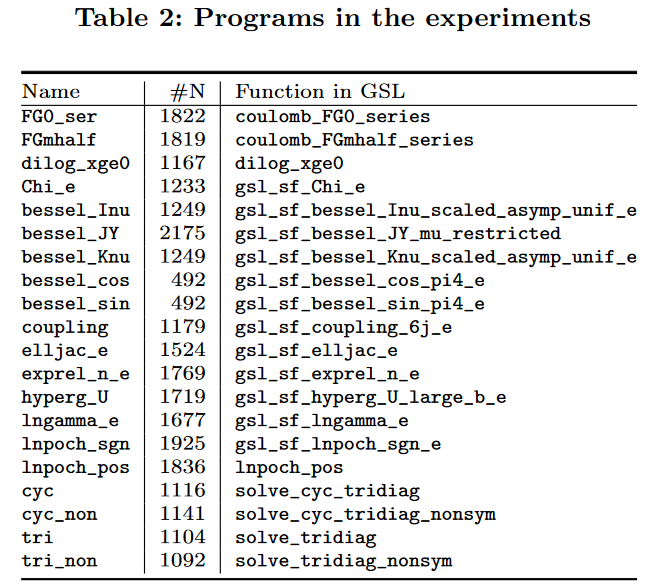
# 5. 评价

为回答RQ4，我们先用我们的贪婪算法与用随机生成的马尔科夫模型（节3.3的）进行性能比较。即，我们运行贪婪算法，假定我们知道求解约束的代价但不知道迁移概率，并测量它的性能。换句话说，迁移概率如算法1中在运行时被估计出来。结果见表1最后一行。能观察出，与现有启发式方法相比，贪婪算法效果更好。

|  |
| --- |
| RQ3的答案：设计一个实际存在的逼近最优策略的算法是可能的。 |

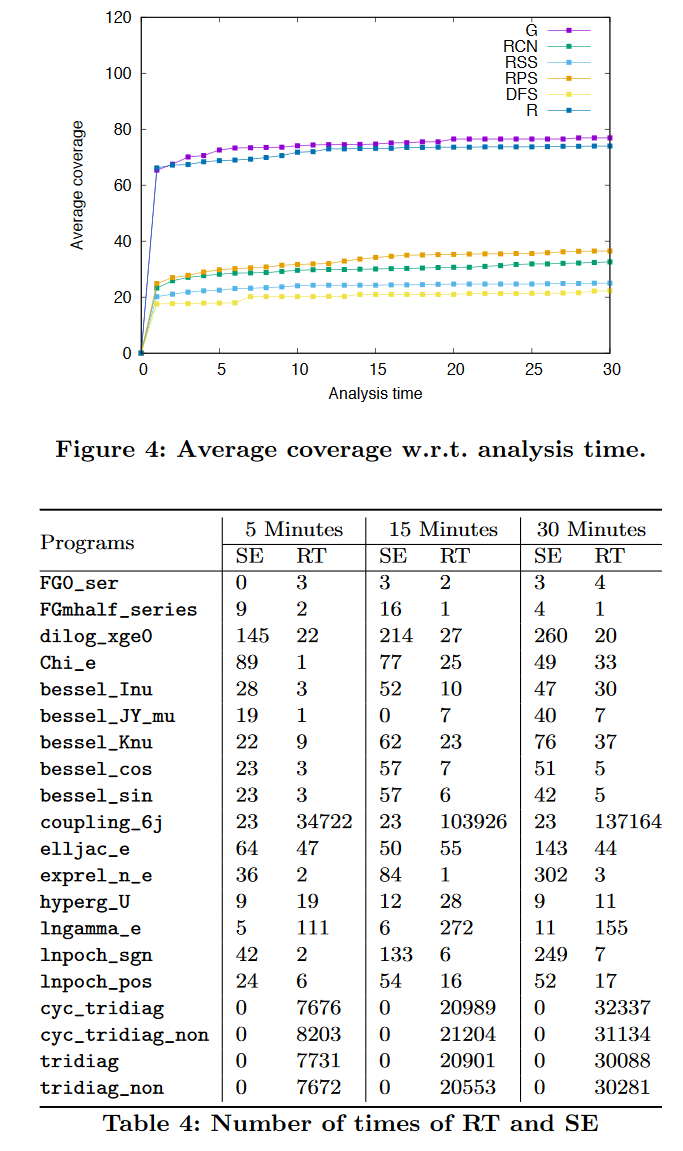
至今我们已经在抽象模型上实验了不同策略。为基于真实世界的程序解答RQ4, RQ5，我们基于KLEE实现我们的方法。注意KLEE是一个符号执行引擎，即它只含符号变量的符号值。我们因此首先用每个变量既含符号值又含有具体值的具体-符号执行引擎扩展KLEE。就结果而言，我们能做到贪婪算法中在运行时在随机测试与符号执行间切换（如果有必要）。在（符号或具体）执行时，当遇到一个分支陈述，我们派生一个和未被选择分支相对应的状态，无需考虑他的可行性。

为了估计，我们先构建过程间控制图（ICFG），它的节点是的状态。如算法1，迁移概率在运行时被估计。记得我们需要求解一个方程组来选择局部最优的测试生成法。在我们的实现中，我们用Eigen[16]求解方程组。当选择符号执行，求解出符号状态的路径约束检查状态的可行性。如果状态不可行，标记路径为不可行。如果选择随机测试，我们生成一个带有具体值的随机测试用例。



实验设置：为验证不同具体-符号执行策略的有效性，我们需要一套程序，包括复杂的路径条件（这样的话约束求解路径条件需要花不平凡的时间量）还有不平凡控制流（这样的话不同策略可以选择不同的测试生成方法和路径）。我们用GNU科学库（GSL）的程序[25]。GSL的函数经常同时具有复杂的算术运算与控制流。GSL先前已经用KLEE分析过了[29,39]。我们对GSL中的函数进行排序，使用之前研究中针对每个函数实现的代码覆盖率，并选择覆盖率最低的函数。我们实验中使用表2列出的函数，其中第二列表示函数的ICFG中节点数（也就是函数或被调用函数中基础块的数目）。注意我们过滤掉带有如函数指针、复杂数组和复杂结构体的复杂输入类型的函数。为分析这些程序，我们为分析浮点程序已经在[40]实现了方法。基本思想是转换浮点操作为整型模拟函数。我们在我们的实现中如库中使用软浮点[43]。此外，约束下的符号执行[17]也被实现，用来分析任意函数。

为基准比较，我们用我们的方法对比3.3节解释的KLEE中支持的4个搜索启发式方法。注意实现其他搜索策略的工具，如[23,33,42]，既不维护也不针对不同的编程语言。用6个策略分析每个函数，我们测量通过不同的超时达到的instruction coverage。我们的实验在一个有64GB内存、16核3.2GHz XEON CPU的服务器上开展的。每个求解约束的超时是10秒。求解约束的代价用[29]中的公式估计。每个实验重复三次，我们报告平均结果。我们的实现和程序在[30]中有。



评价结果：表3展现超时分别为5分钟、15分钟和30分钟每个函数达到的覆盖，其中列G是我们的贪婪算法的结果，R是总是用随机测试策略。每种设置的胜者用粗体高亮。注意由于随机性，并不保证多次测试总能达到更好覆盖。能从我们的贪婪算法看出，不但达到更好的覆盖，而且更快。例如，5分钟之后，我们的贪婪算法覆盖72.6%的指令，而所有KLEE策略的覆盖减少了30%。30分钟后，我们的覆盖是76.9%，而KLEE的策略减少37%。注意随机测试也能达到比KLEE更好的效果（虽然还是比我们的差）。结果说明现存的具体-符号测试策略应该更好地整合随机测试，而且我们的策略提供达成这一点的有效方法。

图4可视化了每个策略随着时间的覆盖趋势。能观察出1分钟内，我们的贪婪算法能达到一个比KLEE更高的覆盖。这是因为我们能每次都有策略地选择奖励最高的方法，并快速覆盖那些容易覆盖的指令。随后，我们的策略通过求解很难求解的路径条件或生成随机测试用例，缓慢地获得更多覆盖。大体上，其他策略也在一分钟后有轻微的覆盖增长，虽然并不总是这样。在少数用例中，如dilog\_xge0，在RPS策略15分钟超时中能观察到一个覆盖中的增长。这可能是因为该策略转换到解决不同路径，从而导致许多未发现的指令。

为了了解我们的策略对测试用例生成方法的选择，我们总结 4中随机测试和符号执行的应用次数。注意到表中的每个条目对应不同的有不同超时的运行，因此不能保证30分钟列中的数字是最大的。能看出我们的策略并不总是比一种方法更倾向于另一种方法。此外，随机测试的次数与符号执行的次数之比在函数之间存在显著差异。例如最后四个函数，由于复杂的路径条件，一直选择随机测试，结果有效达到高覆盖。函数exprel\_n\_e表现出另一个极端，也就是符号执行因为它的简单约束而成为常见选择。这说明我们的策略适应于不同的函数。

|  |
| --- |
| RQ5的答案：我们的贪婪算法比KLEE中现存的启发式方法表现更好。 |

效度威胁：我们的测试对象都是数值函数，因此结果可能有偏差。我们计划将所提议的方法应用于其他程序（例如，在非平凡数据结构上运行的程序），以进一步验证其总体效果。

# 6. 相关工作

这项工作与许多现有的具体-符号测试搜索策略密切相关。除了前面提到的那些，还有其他搜索方法。在[14]中，刘某等人提出了以经验性地预测解决路径约束的代价，并以较小的求解代价对这些路径进行优先级排序。在[38]，Park等人提出了CarFast策略，该策略总是选择一个其相对且尚未覆盖的分支，并且在这个分支上具有最多的语句控制依赖。谢等人。 [48]提出了用适应性引导的路径探索的技术，该技术通过计算执行路径和分支的适应值，来指导下一次朝特定分支的执行。适应度函数计算一个已覆盖的路径与另一个尚未覆盖的分支的接近程度。Marinescu等人。[36]提出了对软件补丁的符号执行。它利用提供的测试组件来识别一个好的测试用例，并使用带有几种试探方式的符号执行来生成更多相关输入来测试补丁。在[42]，Seo等人提出了上下文引导的搜索策略，它在下一轮输入生成的新上下文（即，局部分支选择序列）下选择分支。在[8]中，Cadar等人使用了最佳优先搜索策略，该策略检查所有执行状态并迫使符号执行朝向危险的操作（例如，取消引用指针）。与上述方法相比，我们是第一个正式定义什么是最优策略并随后开发实用算法的方法。我们提供了一个框架，用于系统地比较随机测试和符号执行的有效性。

这项工作与结合随机测试和符号执行的工作有关。除了[3,4]，我们也已在第1节中讨论过，Kong等人 [31]讨论了在验证混合自动机的过程中将随机测试和符号执行相结合的不同策略。他们也在选择应用符号执行的位置时利用了转移概率和代价。然而，他们的方法仍然是试探性的（即，选择具有低代价的分支，类似于[14]中的方法），因为没有最优策略的定义。混合具体-符号测试[35]结合了随机测试和具体-符号测试。我们的想法是从随机测试开始，通过执行大量随机测试用例来快速达到程序的深层。当随机测试停止扩大覆盖范围一段时间后，它会切换到具体-符号测试，以便从当前程序状态彻底搜索状态空间。Garg等人。[20]提出将反馈导向的单元测试生成方法与经典测试相结合。他们从类似于Randoop [37]的随机单元测试开始，并在单元测试覆盖平台时切换到进行具体-符号测试。在[49]中提出了类似的想法。与上述方法相比，我们的方法正式分析了随机测试和符号执行的有效性，并使得我们在每次迭代中可以选择更有效的方法。

这项工作与通过修剪路径[1,5,10,24,28]和并行性[44]等方法降低符号执行和具体-符号测试代价的工作有很大关系。

# 7. 结论

在这项工作中，本文提出了一个框架来推导出最优的具体-符号测试策略，在此基础上我们分析现有的启发式方法，并提出一种新算法来逼近最优策略。对随机生成的模型和一组真实C程序的评估表明，我们的算法经常优于大多数现有的基于启发式的算法。

对于未来的工作，我们希望研究估算概率和解决程序路径成本的替代方法。此外，我们希望将我们的框架扩展到其他测试用例生成方法。

# 参考文献

[1] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demanddriven compositional symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS*, pages 367–381, 2008. [2] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

1. Marcel Bo¨hme and Soumya Paul. On the efficiency of automated testing. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22)*, pages 632–642, 2014.
2. Marcel Bo¨hme and Soumya Paul. A probabilistic analysis of the efficiency of automated software testing. *IEEE Trans. Software Eng.*, 42(4):345–360, 2016.
3. Peter Boonstoppel, Cristian Cadar, and Dawson R. Engler. Rwset: Attacking path explosion in constraint-based test generation. In

*Tools and Algorithms for the Construction and Analysis of*

*Systems, 14th International Conference, TACAS 2008, Held as*

*Part of the Joint European Conferences on Theory and Practice of Software, ETAPS*, pages 351–366, 2008.

1. Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *23rd IEEE/ACM International Conference on Automated Software Engineering ASE*, pages 443–446, 2008. [7] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI*, pages 209–224, 2008.
2. Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12(2):10:1–10:38, 2008.
3. Supratik Chakraborty, Dror Fried, Kuldeep S. Meel, and Moshe Y. Vardi. From weighted to unweighted model counting. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI*, pages 689–695, 2015.
4. Ting Chen, Xiaosong Zhang, Shi-ze Guo, Hong-yuan Li, and Yue Wu. State of the art: Dynamic symbolic execution for automated test generation. *Future Generation Comp. Syst.*, 29(7):1758–1773, 2013.
5. Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.
6. Lori A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Software Eng.*, 2(3):215–222, 1976.
7. G Cochran. Laplace s ratio estimator. *Contributions to survey sampling and applied statistics*, pages 3–10, 1978.
8. Leonardo Mendonc¸a de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and*

*Analysis of Systems, 14th International Conference, TACAS*

*2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS*, pages 337–340, 2008.

1. Matthew B. Dwyer, Antonio Filieri, Jaco Geldenhuys, Mitchell J. Gerrard, Corina S. Pasareanu, and Willem Visser. Probabilistic program analysis. In *Grand Timely Topics in Software Engineering - International Summer School GTTSE*, pages 1–25,

2015.

1. Eigen 3.3.4. Eigen Website. [http://eigen.tuxfamily.org/.](http://eigen.tuxfamily.org/)
2. Dawson R. Engler and Daniel Dunbar. Under-constrained execution: making automatic code destruction easy and scalable. In *Proc. ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2007)*, pages 1–4. ACM, 2007.
3. Antonio Filieri, Marcelo F. Frias, Corina S. Pasareanu, and Willem Visser. Model counting for complex data structures. In *Model*

*Checking Software - 22nd International Symposium, SPIN*, pages 222–241, 2015.

1. William A Gale and Geoffrey Sampson. Good-turing frequency estimation without tears\*. *Journal of Quantitative Linguistics*, 2(3):217–237, 1995.
2. Pranav Garg, Franjo Ivancic, Gogul Balakrishnan, Naoto Maeda, and Aarti Gupta. Feedback-directed unit test generation for C/C++ using concolic execution. In *35th International Conference on Software Engineering, ICSE*, pages 132–141, 2013.
3. Jaco Geldenhuys, Matthew B. Dwyer, and Willem Visser. Probabilistic symbolic execution. In *International Symposium on Software Testing and Analysis, ISSTA*, pages 166–176, 2012.
4. Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223, 2005.
5. Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium, NDSS*, 2008.
6. Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and SaiDeep Tetali. Compositional may-must program analysis: unleashing the power of alternation. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, pages 43–56, 2010.
7. GSL 2.1. GNU Scientific Library (GSL). [http://www.gnu.org/ software/gsl/.](http://www.gnu.org/software/gsl/)
8. Richard G. Hamlet. Testing programs with finite sets of data.

*Comput. J.*, 20(3):232–237, 1977.

1. Ronald A. Howard. The M.I.T. Press, 1960.
2. Joxan Jaffar, Vijayaraghavan Murali, and Jorge A. Navas. Boosting concolic testing via interpolation. In *Joint Meeting of the European Software Engineering Conference and the ACM SIG-*

*SOFT Symposium on the Foundations of Software Engineering, ESEC/FSE’13*, pages 48–58, 2013.

1. Liu Jingde, Chen Zhenbang, and Wang Ji. Solving cost prediction based search in symbolic execution. *Journal of Computer Research and Development*, pages 1086,1094, 2016.
2. Sun Jun. http://sav.sutd.edu.sg/research/smartconcolic.
3. Pingfan Kong, Yi Li, Xiaohong Chen, Jun Sun, Meng Sun, and Jingyi Wang. Towards concolic testing for hybrid systems. In *FM*

*2016: Formal Methods - 21st International Symposium*, pages 460–478, 2016.

1. Marta Kwiatkowska, Gethin Norman, and David Parker. Prism: Probabilistic symbolic model checker. In *Computer performance evaluation: modelling techniques and tools*, pages 200–204. Springer, 2002.
2. You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. Steering symbolic execution to less traveled paths. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA*, pages 19–32, 2013.
3. Kasper Søe Luckow, Marko Dimjasevic, Dimitra Giannakopoulou, Falk Howar, Malte Isberner, Temesghen Kahsai, Zvonimir Rakamaric, and Vishwanath Raman. Jdart: A dynamic symbolic analysis framework. In *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS*

*2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS*, pages 442–459, 2016.

1. Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *29th International Conference on Software Engineering (ICSE*, pages 416–426, 2007.
2. Paul Dan Marinescu and Cristian Cadar. High-coverage symbolic patch testing. In *Model Checking Software - 19th International Workshop, SPIN*, pages 7–21, 2012.
3. Carlos Pacheco and Michael D. Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd Annual*

*ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, pages 815–816, 2007.

1. Sangmin Park, B. M. Mainul Hossain, Ishtiaque Hussain, Christoph Csallner, Mark Grechanik, Kunal Taneja, Chen Fu, and Qing Xie. Carfast: achieving higher statement coverage faster.

In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE’12, Cary, NC, USA - November 11 - 16, 2012*, page 35, 2012.

1. Minghui Quan. Hotspot symbolic execution of floating-point programs. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE*, pages 1112–1114, 2016.
2. Anthony Romano. Practical floating-point tests with integer code. In *Proc. International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2014*, pages 337–356. Springer, 2014.
3. Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM*

*SIGSOFT International Symposium on Foundations of Software Engineering*, pages 263–272, 2005.

1. Hyunmin Seo and Sunghun Kim. How we get there: a contextguided search strategy in concolic testing. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22)*, pages 413–424, 2014.
2. SoftFloat 2b. Berkeley SoftFloat. [http://www.jhauser.us/ arithmetic/SoftFloat.html.](http://www.jhauser.us/arithmetic/SoftFloat.html)
3. Matt Staats and Corina S. Pasareanu. Parallel symbolic execution for structural test generation. In *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA*, pages 183–194, 2010.
4. Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *23rd Annual Network and Distributed System Security Symposium, NDSS*, 2016.
5. Deian Tabakov and Moshe Y. Vardi. Experimental evaluation of classical automata constructions. In *Logic for Programming, Artificial Intelligence, and Reasoning, 12th International Conference, LPAR*, pages 396–411, 2005.
6. Nikolai Tillmann and Jonathan de Halleux. Pex-white box test generation for .net. In *Tests and Proofs, Second International Conference, TAP*, pages 134–153, 2008.
7. Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN*, pages 359–368, 2009.

Chaoqiang Zhang, Alex Groce, and Mohammad Amin Alipour. Using test case reduction and prioritization to improve symbolic execution. In *International Symposium on Software Testing and Analysis, ISSTA*, pages 160–170, 2014.