arXiv:2601.12056v1 [cs.CC] 17 Jan 2026

外延定义场景中自适应测试的复杂度*

IsmaelRodríguez[1,3], DavidRubio[2],和 Fernando Rubio[1,3] †‡§

摘要

本文考察一种测试场景：被测实现（IUT）的可能定义集合，以及这些定义在所有可能交互中的行为，都是以外延方式定义的，即逐个元素、逐个案例进行定义。在此场景下，我们研究如何找到最小测试策略，使得收集到的观测结果必然足以判断被测实现是否正确（即它是否必然属于可能正确的定义集合），并考虑四种问题变体：有无非确定性，以及可能正确和错误的定义集合中是否都可能包含多个定义。本文研究了这些变体的计算复杂性，并确定了 PSPACE 完全性和 Log-APX 困难性等性质。

关键词： 形式化测试、自适应测试、计算复杂性、PSPACE 完全性、近似困难性。

1 引言

形式化测试技术（FTT）的研究提出了大量模型，用于探讨许多不同可能场景下的测试。在形式化测试模型中，可以考虑影响测试活动的诸多方面：例如，可以假设被测实现（IUT）是用确定性的或非确定性的有限状态机（FSM） [1, 2, 3], 输入/输出标记迁移系统 [4, 5, 6], 时序自动机 [7,8, 9, 10], 概率系统 [11, 12, 13], 等语言编写的；可以假设被测实现相对于期望行为能够暴露的偏差集合由

*本文发表于《计算机科学前沿》。当前版本为作者已接受的稿件。

†1计算机系统与计算系，信息学院，马德里康普顿斯大学，西班牙马德里 28040。

‡2瓦伦西亚生物力学研究所。瓦伦西亚理工大学，西班牙瓦伦西亚 46022。§3知识技术研究所，马德里康普顿斯大学，西班牙马德里 28040。

例如，故障模型 [14], 一组关于被测实现的假设 [15, 16, 17],等给出；我们与被测实现交互或收集其响应的方式也可以是例如通过延迟交互、不精确观测 [18, 19], 等。

后一类测试中一个特别值得关注的问题是：对被测实现施加的测试用例，必须在测试活动开始前确定（称为预设测试），还是可以根据此前施加输入或测试用例时收集到的观测结果（输出），动态决定下一步施加的输入（称为自适应测试） [1, 20, 21, 22, 23, 24]。后一种情况下，较少的测试工作或许就能获得更多关于被测实现是否正确的信息，因为可以利用先前的观测结果来引导后续交互，使被测实现进入能够更敏锐地揭示其正确性与否的状态。

人们已针对不同框架开发自适应测试策略。通常，这些策略取决于被测实现所采用的描述语言。例如，有限状态机的自适应测试研究尤为活跃 [1, 20], ，尽管近年来标记迁移系统也受到了关注 [25], ；此外，人们还根据时序逻辑规约推导出了自适应测试策略 [26]。自适应测试方法也取决于测试者的目标，例如，确保一致性测试 [23, 24], 、构造引导序列或同步序列 [20, 22, 21],等。尽管存在这些差异，只要足够抽象地看，所有方法在设计自适应策略时面临的挑战本质上都是相同的：我们需要动态选择测试者的下一步操作，从而提高后续观测所提取信息的重要性。事实上，自适应测试通常可以视为测试者与被测实现之间的一场博弈：测试者力求揭示被测实现故障，而被测实现则力求掩盖故障。当然，被测实现并不会有意做任何事；但要发现被测实现故障的策略无论如何都必须涵盖被测实现（无意中）特别善于隐藏故障的情形。因此，针对这些最坏情况进行推理就仿佛被测实现能够在有限的自由度内决定展示什么，以增加测试者的难度。所有自适应测试场景共有的这一目标，能否以一种一般的方式加以定义和研究，也就是说，不依赖于特定的被测实现语言、特定的故障形式或测试活动中的任何其他具体难点?

近年来，一些研究工作从非常一般且抽象的角度提出了用于分析测试的模型，而不论被测实现所采用的定义语言、所假设的故障模型或假设，还是特定的一致性判据。这些工作包括一般性的可测试性模型，其度量标准是测试套件达到完备性所需的基数 [27,18], ，尤其是它们究竟是有限的、可数无限的、不存在的，等等；同时也包括测试复杂度，其度量标准是随着所应用测试用例数量的增加，我们对被测实现正确性的确信程度提升的速度 [19]。尽管这些模型具有很强的一般性，但它们并未将测试视为一个后续操作取决于先前观测结果的过程，而这正是自适应测试的本质。在本文中，我们采用一个旨在捕捉自适应测试本质的简单模型，研究自适应测试的计算复杂性，

并将其置于最基本的层级。事实上，我们并不提供一个具有足够一般性的模型来完整表达此前关于自适应测试的任何模型；相反，我们把握自适应测试一般意义上的本质难点的方式恰恰相反：我们将研究它最基础、最简单、最本质且不那么紧凑的形式，即这样一种情形：被测实现针对所有可能交互的可能行为集合是外延定义的。

这种 外延定义 意味着，我们假设（黑盒）被测实现的实际定义属于给定的有限个可能的被测实现定义集合，并且这些定义逐一进行了描述。因此，这种观点与以下做法相反：将被测实现的可能定义集合紧凑地刻画为所有满足某条给定规则的定义，例如“被测实现属于这样一组有限状态机：它们按照某个给定的规范有限状态机定义，但至多存在一个输出故障”或“被测实现是代码行数不超过 *1,000* 行的任意 *Java* 程序”。此外，我们的外延定义观点始终一致地应用于每个可能的被测实现定义本身，因为与每个可能的被测实现定义进行交互的所有可能方式也都逐一给出了描述。因此，假设这些交互的集合是有限的；相应地，分别发送给被测实现或从被测实现接收的可能输入和输出集合也都是有限的。假设对被测实现的所有输入应用彼此独立，因此，被测实现在每次输入后都能可靠地返回其初始配置。这样，输入可以视为测试用例，而输出则可以视为这些测试用例的观测结果。[1]

因此，我们通过外延方式描述被测实现（IUT）在所有情形下的行为：列出其所有可能的完整行为定义，其中每个定义都列出了在所有可能输入下可能出现的所有响应（即输出），通常这些响应具有非确定性。此外，在被测实现所有可能定义的集合中，外延定义的一个子集表示根据测试者的要求被视为正确的那些定义。测试者的目标是找出一种自适应测试策略，以判定被测实现是否正确；如果存在这样的策略，最好能找到最短的策略。通过自适应策略判断被测实现是否正确，意味着向被测实现提供某个输入，使得无论被测实现返回什么输出，测试者都能据此继续提供某个输入——通常会根据先前输出的不同而有所不同；之后，对于被测实现返回的每个输出，测试者都能继续提供输入，如此反复，最终在所有情况下都能将被测实现判定为必然正确（即属于前述子集），或必然错误（即不属于该子集）。

尽管前一个模型表面上过于具体，但它之所以特别有趣，有两个原因。一方面，请注意，研究问题最具体且简单版本的计算复杂性下界要有趣得多，因为证明问题 A 在某个复杂性类上是困难的，会直接推出将问题 A一般化的任何问题 B 在该复杂性类上也是困难的。因此，对于任何能够直接表示我们的外延定义模型的模型（例如，

[1]这意味着，在该模型中，必须将连续机器输入的一个序列 $i_1 \dots i_n$ 应用于被测实现的过程抽象为一个单一输入 σ，以表示整个交互序列 $i_1 \dots i_n$。由于假定输入和输出构成有限集，因此无论如何，都将考虑与被测实现进行交互的有限个可能交互序列 σ_1，……，σ_m。

更具表达力的 FSM 模型可以轻松用来表示我们的外延模型），其自适应测试的困难性至少也会达到我们针对外延模型所发现的程度。我们将证明，在外延定义的设置下，自适应测试的一些变体具有 PSPACE 和 Log-APX 等复杂性类中的复杂性困难性。需要回顾的是，在该模型中，假定输入应用彼此独立，因此这些困难性结果实际上是针对无记忆模型得到的。这说明了一点，乍看之下可能令人惊讶：任何计算模型固有的状态并不是使自适应测试成为困难任务所必需的，甚至不需要使其成为 PSPACE 困难问题。

另一方面，值得注意的是，在许多软件工程场景中，测试者可对被测实现执行的测试用例是外延定义的：测试者不会从给定的、表达能力丰富的规范模型以及可能形式化定义的故障模型中自动提取测试用例——这构成了一种理想的测试方法——而是手动定义有限集测试用例，并为每个测试用例规定被测实现可能给出的响应，其中每个错误响应都表示被测实现可能存在一种故障。由于无论如何都不会测试被测实现在这些手动定义的测试用例未涵盖的交互中的响应，因此这些响应不会向测试者提供任何信息，所以，这种模型等同于逐项列出被测实现有限集可能存在故障的定义，并针对每一种定义，规定每个可能测试用例对应的有限集可能响应。这种务实而简单的模型，或者粗略地说，这种缺乏模型的做法，最终在测试从业者中十分常见，结果证明它与我们的外延定义模型等价。事实上，对许多从业者而言，规范不过是所有遵循全部需求的可能定义所构成的集合，而故障模型不过是根据他们明确认为可能发生的错误，系统可能存在的所有错误定义所构成的集合。在这些情况下，两者的性质按定义由外延定义的集合给出：列表中的一切都是可能的，列表之外的一切都不可能——而测试数据之间任何可能的类比，恰恰源自这些定义中的规律性。

我们将在外延模型下研究自适应测试的计算复杂性，考察由两种二元选择组合而成的四种不同问题变体。一方面，我们将考虑是否允许非确定性。另一方面，我们将考虑以下两种情况：被测实现（IUT）的可能正确的定义子集或错误定义子集为单元素集合，或者这两个集合都包含多个元素。[2] 我们将确定由此产生的四个问题的计算复杂性，并得出它们在不同复杂度类中的复杂性困难性：要求确定性的两种情况都是 NP 完全，其中一种是 Log-APX 完全，另一种至少在 Log-APX 中是困难的；涉及非确定性的两种情况

[2]我们假设，一旦观测结果使我们能够排除所有可能正确的被测实现定义，或排除所有可能错误的被测实现定义，测试过程就会立即停止——无论另一类定义是否仍有多个可能选项。请注意，分别而言，该条件足以提供明确的错误性诊断或正确性诊断。我们的目标并不是在可能性集合中唯一地识别被测实现定义，因为这将需要更强的条件。

都是 PSPACE 完全。请注意，除非 P = NP，否则不存在多项式时间算法能够为前两种问题求得精确解。此外，鉴于这些问题在 Log-APX 中的困难性，已知除非 P =NP，否则不存在多项式时间近似算法能够保证任何常数性能比；也就是说，无法保证返回解的质量与最优解质量之比有一个常数上界。对于后两种问题，即允许非确定性的情况，它们在 PSPACE 中的困难性意味着，除非 P = PSPACE，否则不存在多项式时间算法能够解决这些问题。鉴于 P $\subseteq$ NP $\subseteq$ PSPACE，这一性质比 P = NP 更强，也更难以预期。这些复杂性困难性结果还有其他实际意义：我们可以使用遗传算法等快速的次优元启发式方法来处理 NP 完全问题，而对于 PSPACE 完全问题，则应考虑极小极大算法等成本更高的解决方案。

本文其余部分的组织如下。下一节将通过一个动机示例，非正式地介绍我们设定的关键方面。第 3 节将正式介绍模型，并给出和证明复杂度结果。第 4 节将给出结论与未来工作。论文附录中展示了定理证明的详细过程。

此外，尽管本文旨在进行理论研究，因为我们正在确定所研究问题的计算复杂性，论文附录中也给出了一些关于尽管这些问题复杂度较高、如何在实践中处理它们的思考。附录中概述了一个比下一节所给示例复杂得多的示例，并利用该示例介绍这些实践难题，同时讨论如何应对这些难题。

2 动机示例

本节使用机器人系统的一个简化版本，介绍我们测试设定中的主要概念。让我们测试一台在可能不稳定且不平衡的表面上行驶的自主四轮探测漫游车的碰撞避免系统 （CAS）。CAS 中考虑了三种可能的交互情况：

A.- 如果预计碰撞物体会在不到一秒内被触及，即近处碰撞，那么机器人会完全转向以避开它。假设这一操作存在一定风险，但由于机器人距离碰撞物体太近，无法在碰撞前完全制动，因此认为这是最佳选择。B.- 如果预计碰撞物体会在两秒以后才被触及，即远处 碰撞，那么机器人会制动，在碰撞发生前完全停下。在这种情况下，不选择转向这一备选方案，因为与停车相比，转向被认为会带来不必要的风险。C.- 如果预计碰撞物体会在一到两秒内被触及，即中间 碰撞，那么机器人可以选择

$f_1(near) = \{turn\}$
$f_1(far) = \{brake\}$
$f_1(inter) = \{turn, brake\}$

$f_2(near) = \{turn\}$
$f_2(far) = \{brake\}$
$f_2(inter) = \{turn\}$

$f_3(near) = \{turn\}$
$f_3(far) = \{brake\}$
$f_3(inter) = \{brake\}$

$f_4(near) = \{turn\}$
$f_4(far) = \{brake\}$
$f_4(inter) = \{both\}$

$f_5(near) = \{turn\}$
$f_5(far) = \{nothing\}$
$f_5(inter) = \{turn\}$

$f_6(near) = \{nothing\}$
$f_6(far) = \{brake\}$
$f_6(inter) = \{brake\}$

$f_7(near) = \{turn\}$
$f_7(far) = \{turn\}$
$f_7(inter) = \{turn\}$

$f_8(near) = \{brake\}$
$f_8(far) = \{brake\}$
$f_8(inter) = \{brake\}$

$f_9(near) = \{nothing\}$
$f_9(far) = \{brake\}$
$f_9(inter) = \{nothing\}$

$f_{10}(near) = \{turn\}$
$f_{10}(far) = \{nothing\}$
$f_{10}(inter) = \{nothing\}$

$f_{11}(near) = \{nothing\}$
$f_{11}(far) = \{nothing\}$
$f_{11}(inter) = \{nothing\}$

$f_{12}(near) = \{turn\}$
$f_{12}(far) = \{nothing\}$
$f_{12}(inter) = \{turn, nothing\}$

$f_{13}(near) = \{nothing\}$
$f_{13}(far) = \{brake\}$
$f_{13}(inter) = \{nothing, brake\}$

图1：被测实现的可能定义。定义 f_1-f_3 被认为是正确的，而 f_4-f_{13} 被认为是错误的。

完全制动或完全转向，因为两者被认为风险程度相近。不过，在这种情况下，机器人不能同时制动并且转向，因为在轮胎因完全制动而被锁住时完全转向，可能导致机器人翻倒。

测试者担心编程团队在开发机器人时可能犯了一些错误，因此将机器人部署在实验室中，并观察其在不同预期碰撞场景下的行为，以排除故障。为了测试机器人，测试者将一个物体放置在距离机器人某个特定距离的位置，并使机器人向该物体移动。因此，测试者为激励机器人而产生的可能输入集合（测试用例）就是案例（A）至（C）中提到的那些输入，即 $I = \{near, far, inter\}$。此外，机器人的可能反应集合（输出）定义为 $O = \{nothing, turn, brake, both\}$。根据前述条件（A）至（C）所提供的有限自由度，以及测试者认为程序员可能犯下的错误，构造出被测实现可能的正确和错误完整定义集合，即针对所有所考虑输入的定义。每个完整定义都被形式化描述为一个函数：对于每个可能的输入，该函数返回被测实现接收到该输入后可能作出的输出集合。当该集合为单元素集时，该可能的被测实现定义针对特定输入的行为是确定性的；否则就是非确定性的。

根据需求 (A)-(C)，图1所示的可能 IUT 定义 f_1, f_2, f_3均被视为正确。请注意，定义 f_1 允许在中间距离碰撞（输入$inter$）时以非确定性的方式制动或转向，而需求 (C)允许这样做。还要注意，定义 f_2 和f_3 完全是确定性的。测试者担心实现人员可能犯下以下三种错误：

1.- 在中间距离碰撞中，程序员本可以让机器人同时制动和转向。

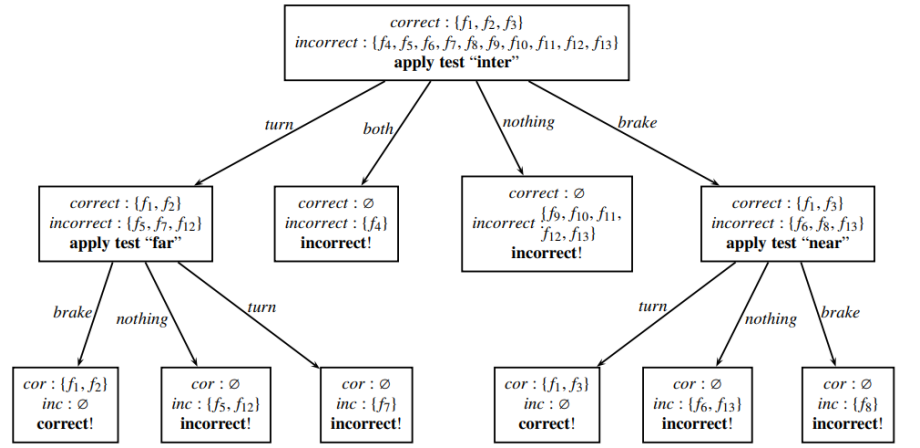


图2：在所有情况下最多应用两个 测试用例的自适应测试策略。

2.- 程序员可能错误地实现了CAS信号请求，要求机器人制动，导致它在这种情况下什么也不做。或者，转向信号也可能发生了同样的问题，甚至两种信号都同时如此。

3.- 程序员可能完全忽略了 CAS 规范要求的情形区分，导致机器人在所有情况下都制动。或者，他们也可能在转向操作中犯了同样的错误，而不是在制动操作中。

这些错误可能导致的 IUT 错误定义如图1所示。错误（1）用可能的 IUT 错误定义 f_4 表示。此外，错误（2）用多种错误定义表示，这些定义是通过修改正确的定义 f_1 （f_{12} 和 f_{13}）、正确的定义 f_2（f_5 和 f_9）、正确的定义 f_3 （f_6 和 f_{10}）以及上述任意一种定义（f_{11}）得到的。请注意，错误定义 f_{12} 和 f_{13} 是非确定性的。最后，错误（3）用错误函数 f_7 和 f_8表示。

先前关于可能存在程序员错误的测试人员假设，被形式化为测试人员假设：实际的被测实现定义属于集合$\{f_1, \ldots, f_{13}\}$，该集合由满足需求 (A)–(C) 的定义（即f_1、 f_2、 f_3）以及根据错误 (1)–(3) 不满足这些需求的定义（即 f_4、.....、f_{13}）组成。测试者的目标是与被测实现交互、收集观测结果，并利用这些观测结果判断被测实现是否必然正确（即观测结果足以排除所有可能的错误定义）或错误（观测结果排除了所有可能的正确定义）。假设测试者可以根据先前测试收集到的观测结果决定下一步要执行的测试用例，即采用自适应测试，我们将研究以下问题：在最坏情况下，需要多少次测试才能判断被测实现是否正确?

如图2所示，我们不需要应用全部三个可用的测试用例即可作出判断，因为只要其中两个即可应对任何情况，前提是第二个测试用例巧妙地取决于第一个测试用例之后观测到的结果。首先，我们应用测试用例 $inter$。如果该测试的输出为 $both$，那么被测实现必然错误，因为它必定是 f_4。如果输出为 $nothing$，那么被测实现也必然错误，因为它必定是集合 $\{f_9, \ldots, f_{13}\}$ 中的某个定义。然而，如果输出为 $turn$ 或 $brake$，则接下来必须应用第二个测试用例。一方面，如果输出为 $turn$ ，则接下来应用测试用例 far ，被测实现随后可能给出的三种回复都能让我们判断被测实现是否正确。另一方面，如果测试用例 $inter$ 之后的输出为 $brake$，我们接下来应用测试用例 $near$，同样，三种可能的回复都能提供足够信息，使我们判断被测实现是否正确。请注意，作出此类判断并不要求拥有足够信息来唯一确定哪个函数定义了被测实现的行为。例如，在图2底行第一个和第四个框中所描绘的两种情况下，尽管我们不知道被测实现是 f_1 还是 f_2，或是 f_1 还是 f_3，我们仍然给出“正确”的诊断并结束测试。由于这两对函数均为正确函数，且相应框中可能的错误定义集合为空，因此，要判断被测实现必然正确，没有必要区分这两个可能的正确函数。同样，当存在多个错误定义而没有正确定义时，也没有必要区分<style id='17'>错误定义。底行的第二个和第五个框，以及观察到输出 $nothing$后到达的中间行框，便属于这种情况。还要注意，如果定义是非确定性的，则在应用某个测试用例后，一个定义可能同时存在于多个框中。例如， f_1、 f_{12}和 f_{13} 各自都可能出现在中间行的多个框中。

值得注意的是，任何从测试用例 $near$ 开始的其他自适应测试策略，在最坏情况下都需要应用三个测试用例，因此，图2所示树的每个分支中测试用例的应用顺序确实非常重要。应用 $near$后，如果回复为 $nothing$ （即，被测实现错误），则被测实现必须是 f_6、f_9、 f_{11}或 f_{13} ；如果回复为 $brake$ （错误的被测实现），则它是 f_8 ；否则（即回复为 $turn$ ），它可以是剩余八个函数中的任意一个： f_1、 f_2、 f_3、 f_4、 f_5、 f_7、 f_{10}、 f_{12}。随后，仅应用测试用例 far 还不足以给出正确性诊断：观察到 $nothing$ 和 $turn$ 当然可以做到这一点，但得到 $brake$ 则不行，因为剩余的可能函数将是 f_1、 f_2、 f_3、 f_4，其中有三个正确函数和一个错误函数。因此，继续采用测试用例 far 的策略，并不能让我们在应用两个测试用例后完成测试。我们来看看将 far 替换为 $inter$ 是否也行不通。回想一下，在应用第二个测试之前，函数为 f_1、 f_2、 f_3、 f_4、 f_5、 f_7、 f_{10}、 f_{12}。应用 $inter$后，输出 $both$意味着被测实现是 f_4 （错误的），输出 $brake$ 则意味着被测实现是 f_1 或 f_3 （正确的），但输出 $turn$意味着被测实现是 f_1、 f_2、 f_5、 f_7或 f_{12}，其中有两个正确函数和三个错误函数。

可以通过对称论证说明，任何以测试用例 far 开始的测试策略，在最坏

情况下也需要应用三个测试用例。事实上，图2所示的策略是唯一一种在最坏情况下仅用两个测试用例就能提供（不正确性）诊断的策略，因此它是最优的。

引入一些函数重新定义后，这一策略仍然可以让我们判断被测实现是否正确。例如，我们将 f_7对测试用例 far的行为重新定义为 $f_7(far) = \{turn, nothing\}$，这样一来，这个错误函数就变成了非确定性的。随后，同一自适应测试策略仍然可以让我们判断被测实现是否正确；在底部一行第二个方框的可能错误定义集合中加入 f_7 后，图2所示的示意图仍适用于这一替代情形。请注意， f_7 也 出现在第三 个方框的错误集合中，且这一点保持不变。加入 f_7后，第二个方框中的错误性诊断仍然有效，因为此时我们有零个正确函数和三个错误函数。然而，如果按照以下替代方式使 f_7 变成非确定性的，那么无论采用自适应测试还是预设测试策略，都无法判断被测实现是否（不）正确： $f_7(far) = \{turn, brake\}$。请注意，在这种情况下，错误函数 f_7 将无法与正确函数 f_1 和f_2区分开来，因为无论对被测实现应用多少次， f_7 都可以始终对测试用例 far 返回 $brake$ 。

本文还将研究自适应测试问题的受限版本，具体考虑以下情况：假设所有函数都必须是确定性的，这样只需移除示例中的 f_1, f_{12}, f_{13} ，并采用 f_7 的原始定义即可；假设正确函数集或错误函数集为单元素集合，例如移除 f_2 和 f_3，或移除除 f_7 以外的所有错误定义；或者同时假设两项需求成立，例如移除 f_1, f_3, f_{12}, f_{13} ，并采用 f_7 的原始定义。下一节将研究由此得到的四个问题的复杂度。

为了说明在规模更大、更贴近现实的场景中进行自适应测试所面临的困难，并讨论实践中如何应对这些困难，本文附录概述了一个复杂得多的示例。

3 形式化模型与复杂度结果

首先，我们引入定义集合的概念，它表示被测实现可能具有的所有行为定义所构成的集合，其中每个行为定义都由一个函数表示。我们假设 $\mathcal{P}(A)$ 表示集合 A 的幂集。

定义 1（定义集合）令 I 为输入的有限集，令O 为输出的有限集。对于 I 和 O ，定义集合 C 是函数 f 的有限枚举： $I \longrightarrow \mathcal{P}(O)$，其中对于所有 $i \in I$ ，都有 $f(i) \neq \varnothing$。

如果 $f(i) = \{o\}$，即 $f(i)$ 是单元素集，我们有时将其简写为 $f(i) = o$。如果对于所有 $i \in I$, $f(i)$ 都是单元素集，那么我们称 f 是确定性的。如果所有 $f \in C$ 都是确定性的，那么我们称 C 是确定性的。

我们假设每个函数 f 都是外延定义的。也就是说，对于每个输入 $i \in I$ ，我们都明确地定义其值 $f(i)$。□

C 中的每个函数，都是被测实现对所有输入的一种可能行为的形式化表示：给定一个函数 $f \in C$，$f(i)$表示将输入 $i \in I$ 应用于由 f表示的计算制品后，能够得到的已定义输出集合。由于 $f(i)$是一个集合，f可能表示非确定性行为。例如，在我们的动机示例中，$f_1(inter) = \{$转向，中断$\}$，因此 f_1 是非确定性的，尽管 f_2 是确定性的：$f_2(near) = \{$转向$\}$，$f_2(far) = \{$中断 $\}$，且 f_2(中间) $= \{$转向$\}$。请注意，在此示例中，$C = \{f_1, f_2, f_3, \ldots, f_{13}\}$。也就是说，我们通过枚举集合 C 的所有（有限个）元素来对其进行外延定义。

定义集合用于表示在给定测试场景中所考虑的可能实现的集合。隐含地，定义集合 C 表示测试者对被测实现所作的假设，例如参见 [15]。例如，如果测试者假设被测实现是确定性的，那么C 应仅包含确定性函数；如果被测实现假设被测实现至多会对一个输入产生错误输出，那么 C 中的所有函数都应按此方式定义；等等。

定义集合也用于表示符合规范的实现子集。令 C 表示可能实现的集合，$E \subseteq C$ 表示满足规范的实现集合。测试的目标是与被测实现交互，以便根据收集到的响应，判断被测实现是否确实属于 E。通常，我们依次向被测实现施加一些测试，即一些输入 i_1，i_2，$\ldots \in I$，使得观测结果 $o_1 \in f(i_1)$、$o_2 \in f(i_2)$、.....能够帮助我们给出判定。

定义 2（规范与测试场景）令 C 为一个定义集合。C 的规范是一个集合 $E \subseteq C$。测试场景是一个元组（C，E）。□

若 $f \in E$，则 f 表示一种正确的行为，而 $f \in C \backslash E$ 表示 f是错误的。因此，规范 E 隐含地表示一个正确性准则，而集合 $C \backslash E$ 隐含地定义了测试者所假定的故障模型，因为它由 IUT 可能具有的所有错误行为组成。在任何情况下，根据定义，在我们的模型中，函数的正确性取决于它是否属于 E，其中E 是按函数外延定义的。在我们的动机示例中，$E = \{f_1, f_2, f_3\}$，该集合反映了第 2 节中提及的需求 (A)–(C) 隐含定义的规范。此外，$C \backslash E = \{f_4, \ldots, f_{13}\}$，该集合反映了该节中所考虑的错误 (1)–(3) 隐含给出的故障模型。或者，假设某个规范要求一个接收并输出自然数的程序计算输入值的平方，并假定我们的故障模型认为，每个可能的错误程序P_i，若 $i \in [-10, 10]$，其输出都会比实际平方 i 个单位。于是，$E = \{f\}$，其中 $f\ (x) = x^2$；而 $C = \{g_i \mid i \in [-10, 10]\}$，其中$g_i\ (x) = f\ (x) + i$。

在进行自适应测试时，我们不会用集合 $\mathcal{I} \subseteq I$ 来表示可能的测试套件，因为现在我们可以在观察到应用第一个输入后得到的输出后，再选择第二个输入测试；在得到第二个输出后，再选择第三个输入测试，依此类推。

因此，我们更倾向于讨论测试策略，它们既不是集合，也不是序列。测试策略可以看作一棵树，其中每个节点代表一次测试的应用，每个节点的子节点数等于该输入可能产生的输出数，而必要的正确性（或不正确性）判定则标注在树的叶节点上，如图2所示。因此，从树的根节点到叶节点的每条分支所包含的节点序列，都代表了对被测实现应用的一种可能测试序列（类似于预设测试中的测试套件）；但具体对每个被测实现应用哪条分支，取决于被测实现在测试过程中产生的输出序列。

我们的问题旨在判定是否存在一棵测试树，使其深度（即其最大分支上的输入测试数量）至多为 k，其中 k 是给定的自然数。首先，我们来定义仅涉及确定性函数时的判定问题，也就是说，对于所有 $f \in C$ 和 $i \in I$， $f(i)$ 都是单元素集。

定义 3（确定性自适应测试判定问题）设 C 是关于 I 和 O 的确定性定义集合，$E \subseteq C$ 是一个规范，且 k 是一个自然数。若 $(C,\ E,\ k)$ 满足 D-ATDP，则称：

$$
\begin{gathered}
\exists i_1 \in I\ \forall o_1 \in \{f(i_1) : f \in C\} \\
\exists i_2 \in I\ \forall o_2 \in \{f(i_2) : f \in C,\ o_1 = f(i_1)\} \\
\vdots \\
\exists i_l \in I\ \forall o_l \in \{f(i_l) : f \in C,\ o_m = f(i_m)\ \forall m < l\} \\
\vdots \\
\exists i_k \in I\ \forall o_k \in \{f(i_k) : f \in C,\ o_m = f(i_m)\ \forall m < k\} \\
\{f \in C : o_m = f(i_m)\ \ \forall m \le k\} \subseteq E\ \vee \\
\{f \in C : o_m = f(i_m)\ \ \forall m \le k\} \subseteq C \backslash E
\end{gathered}
$$

给定 $(C,\ E,\ k)$，**确定性自适应测试判定问题** 在于判定 $(C,\ E,\ k)$ 是否满足 D-ATDP。为方便表述，我们也用 D-ATDP 指代该问题。对于 D-ATDP 实例，我们假定 k 以二进制形式给出。

当 E 为单元素集或 $C \backslash E$ 为单元素集时，也就是说，只有一个正确函数或只有一个错误函数，所得的特例问题记为 D-ATDP_1。□

在前面的定义中，满足 D-ATDP 实际上就是寻找一种自适应测试策略，应用不超过 k 个输入。[3] 每个新的可能输入测试层级都取决于被测实现先前产生的输出。此外，对于该策略中从根节点到叶节点的每条输入分支，与该分支中观察到的输出相一致的函数 $f \in C$（即，$\{f \in C :\ o_m = f(i_m)\ \ \forall m \le k\}$）要么全部属于

[3]尽管定义要求恰好有 k 个输入，但显然允许使用更少的输入，因为可以轻易添加填充输入以满足定义，例如重复的输入。

正确函数集合 E，要么全部属于错误函数集合 $C\backslash E$。因此，每条输入分支都会将相应的被测实现判定为正确或错误。此外，由于这些分支涵盖了被测实现可能产生的所有输出，整个测试用例树在判断被测实现是否正确这一任务上实际上是完备的 。

定义好判定问题后，定义相应的优化问题就很简单了；我们要寻找满足 D-ATDP 的最小 k，使得 $(C,\ E,\ k)$ 成立。

定义 4（确定性自适应测试优化问题）设 C 是关于 I 和 O 的确定性定义集合，$E \subseteq C$ 是一个规格说明。给定 $(C,\ E)$，确定性自适应测试优化问题（D-ATOP）是指求解满足 D-ATDP$\}$ 的最小 $\{k:\ (C,\ E,\ k)$。

当 E 是单元素集或 $C\backslash E$ 是单元素集时，我们用 D-ATOP$_1$ 指代 D-ATOP。□

在确定性情形下定义了判定问题和优化问题后，我们可以轻松地对其进行修改，以处理非确定性情形。为此，只需考虑到每个函数得到的输出并不唯一。因此，我们必须考虑相应树的每个层级上所有可能的函数所产生的所有可能输出的并集，如下一个定义所示。

定义 5（非确定性自适应测试判定问题）设C 是关于 I 和 O 的定义集合，$E \subseteq C$ 是一个规范，k 是一个自然数。若 $(C,\ E,\ k)$ 满足 N-ATDP，则称：

$$
\begin{array}{c}
\exists i_1 \in I\ \ \forall o_1 \in \underset{f\in C}{\cup} f(i_1) \\
\exists i_2 \in I\ \ \forall o_2 \in \underset{f\in\{f\in C:\, o_1\in f(i_1)\}}{\cup} f(i_2) \\
\vdots \\
\exists i_l \in I\ \ \forall o_l \in \underset{f\in\{f\in C:\, o_m\in f(i_m)\ \forall m<l\}}{\cup} f(i_l) \\
\vdots \\
\exists i_k \in I\ \ \forall o_k \in \underset{f\in\{f\in C:\, o_m\in f(i_m)\ \forall m<k\}}{\cup} f(i_k) \\
\{f \in C : o_m \in f(i_m)\ \ \forall m \le k\} \subseteq E\ \vee \\
\{f \in C : o_m \in f(i_m)\ \ \forall m \le k\} \subseteq C\backslash E
\end{array}
$$

给定 $(C,\ E,\ k)$，非确定性自适应测试判定问题要求判断 $(C,\ E,\ k)$ 是否满足 N-ATDP。该问题本身也记作 N-ATDP。在 N-ATDP实例中，k 将以二进制形式定义。

当 E 或 $C\backslash E$ 为单元素集时，所得问题记作 N-ATDP$_1$。□

同样，定义相应的优化问题十分简单，如下一个定义所示。

定义 6（非确定性自适应测试优化问 题）设 C 是关于 I 和 O的定义集合，$E \subseteq C$ 是一项规格说明。给定 (C, E)，非确定性自适应测试优化问题（N-ATOP）旨在求得 $\min\{k$，使得 (C, E, k) 满足 N-ATDP$\}$。

当 E 是单元素集或 $C \backslash E$ 是单元素集时，我们将使用 N-ATOP$_1$ 来指代 N-ATOP。□

鉴于在 D-ATDP 和 N-ATDP 的满足条件定义中，E 和 $C \backslash E$ 显然受到同等对待（参见定义 3 和 5），下文讨论 D-ATDP$_1$,、D-ATOP$_1$,、N-ATDP$_1$, 和 N-ATOP$_1$ 的所有问题时，只考虑 E 为单元素集的情形。

定义问题之后，我们给出确定性情形下的以下性质。乍看之下，将 D-ATDP 纳入 NP 类似乎有些奇怪，因为 D-ATDP 的解显然可能呈指数规模（它们是树），而且相对于 D-ATDP 实例的表示大小，这些树的深度显然也可能呈指数级，因为其中的 k 是以二进制形式定义的。

定理 1 我们有：

(a) D-ATDP 是 NP 完全问题。(
b) D-ATDP$_1$ 是 NP 完全问题。(c)
D-ATOP 是 Log-APX 困难问题。
(d) D-ATOP$_1$ 是 Log-APX 完全问
题。

□

证明。 我们先证明（c）。为此，我们需要将一个 Log-APX 困难问题多项式归约到 D-ATOP，并保持 Log-APX 困难性。我们考虑从 最小集合覆盖（Minimum Set Cover，MSC） [29]进行 S-归约[4] 。给定一个集合 $\mathcal{C}$，其中包含若干集合 $\mathcal{C} = \{S_1, \ldots, S_m\}$，满足$\bigcup_{S \in \mathcal{C}} S = \{e_1, \ldots, e_p\}$，MSC 的任务是选取 $\mathcal{C}'$的一个子集 $\mathcal{C}$，使得 $\bigcup_{S \in \mathcal{C}'} S = \{e_1, \ldots, e_p\}$，从而满足 $|\mathcal{C}'|$（即 $\mathcal{C}'$中的集合数量）最小化。将 MSC 归约到 D-ATOP 的 S-归约构造如下。对于如前所定义的 MSC 实例 $\mathcal{C}$，我们按照定义 4 中的相同术语定义一个 D-ATOP 实例，其中每个集合对应一个不同的输入，只有一个正确函数（该函数始终返回输出 0），并且对于子集中的每个可能元素分别定义一个错误函数：

- $I = \{1, 2, \ldots, m\}$
- $O = \{0, 1\}$

[4]在 S-归约中，第二个问题的解可以映射为第一个问题中成本完全相同的解，而且两者的最优值也相同。参见例如 [28]。

- $E = \{g\}$，其中 $\forall i \in I\ g(i) = 0$
- $C = E \cup \{f_e\colon\ e \in \{e_1, \ldots, e_p\}\}$ 其中

$$f_e(l) = \begin{cases} 1 & \texttt{if } e \in S_l \\ 0 & \texttt{otherwise} \end{cases}$$

请注意，每个输入测试都代表原始 MSC 问题中的一个集合。此外，原始集合中每个可能的元素都对应一个错误函数；将此函数应用于某个输入（即一个集合）时，当且仅当该元素属于相应的集合，函数才返回 1。

可以看出，当且仅当存在一个代价为$q \in \mathbb{N}$（设其解为 $\{S_{l_1}, \ldots, S_{l_q}\}$）的 D-ATOP 实例解时，原始 MSC 实例也存在一个代价相同的解 q；而且我们可以轻松地将前者映射为后者。

考虑到我们的归约，

$\bigcup_{j=1}^{q} S_{l_j}$ $\mathcal{C}$ $\forall e \in$是一个集合覆盖，$\bigcup\mathcal{C}$ $\exists r \in \{1 \ldots q\}$ $e \in S_{l_r}$ $f_e\ l_r\ = 1$ 当且仅当.....。因此，（.....）。于是，在 D-ATOP 中使用输入l_r，我们可以将区分函数 f_e 与唯一的正确函数 g区分开来；后者始终返回 0，也就是说，观察到的输出总能让我们排除其中一个。因此，输入集合 $\{l_j\colon\ j \in \{1, \ldots, q\}\}$ 能够将 g与 $C\backslash\{g\}$中的所有函数区分开来。由此，按任意顺序应用这些 q 输入后，观察到的输出总能让我们知道被测实现是正确还是错误。

类似地，如果某些 q输入 $\{l_j\colon\ j \in \{1, \ldots, q\}\}$ 能够将g 与所有错误函数区分开来，那么对于每个错误函数 f_e，我们都有一个输入 l_r，使得 $f_e(l_r) = 1$，因为正确函数总是返回 0。因此，在相应的 MSC 问题中，对于每个元素$e \in \bigcup\mathcal{C}$，我们也都有一个集合 S_{l_r}，使得 $e \in S_{l_r}$。于是，$\{S_{l_j}\colon\ j \in \{1, \ldots, q\}\}$ 是 $\mathcal{C}$ 的一个集合覆盖，其代价为 q。

我们得出结论：存在一个从 MSC 到 D-ATOP 的 S-归约，并且该归约证明了 D-ATOP 是 Log-APX 困难的。此外，请注意，我们的 S-归约中提出的映射还将 MSC 多项式归约到 D-ATDP，这证明了 D-ATDP 是 NP-困难的。为了得出 (a)，我们还需要证明 D-ATDP 属于 NP。

让我们回顾一下将测试策略视为树的观点，如图2所示。由于 D-ATDP 中的所有函数都是确定性的（与图2所示的情况相反，图2实际讨论的是 N-ATDP），因此对于测试策略中的任意节点，其子节点处可能出现的所有（正确和错误）函数集合必须两两不相交。通过简单的归纳推理可知，这意味着策略所有叶节点处可能出现的函数集合也必须两两不相交；事实上，这些集合必须构成定义集合 C 的一个划分。令 $n = |C|$ 表示 C 中函数的数量，令 $m = |I|$ 表示输入的数量。由此可知，D-ATDP 的测试策略的叶节点数量不能超过 n。考虑这样一种测试策略：所有应用的输入都不同于同一分支中出现的其他输入，即策略的任何分支都不包含重复输入。显然，该策略的深度不能超过 m。请注意，判定 D-ATDP 无须探索任何包含重复输入的分支，因为重复输入毫无用处。鉴于 D-ATDP 中所有函数的确定性，这一点显而易见。因此，无论 D-ATDP 实例中给定的 C 的值为何（$n = |C|$，n，m），我们总能通过考察叶节点不超过 个且深度不超过 的树来判定 D-ATDP。

包含重复输入，因为重复输入毫无用处。鉴于 D-ATDP 中所有函数的确定性，这一点显而易见。因此，无论 D-ATDP 实例中给定的 k 的值为何（C， E， k），我们总能通过考察叶节点不超过 n 个且深度不超过 m 的树来判定 D-ATDP。由此可知，为判定 D-ATDP 而可能需要考察的任意候选树，其节点总数不能超过 $n \cdot m$。此外，由于 D-ATDP 实例中 C 的表示大小不可能小于 n，也不可能小于 m，因此我们推知，D-ATDP 的候选解（即测试策略）的规模相对于相应问题实例的规模是多项式的。鉴于这一限制，不难看出，检查任意候选解是否有效都能在多项式时间内完成，因此 D-ATDP 属于 NP。

需要指出的是，在我们之前的 S-归约中，得到的 D-ATOP 实例实际上也是 D-ATOP$_1$ 实例，而且两个优化问题中需要最小化的成本函数相同。因此，该 S-归约也证明了 D-ATOP$_1$ 是 Log-APX 困难的，而 D-ATDP$_1$ 是 NP-困难的。

既然我们已经证明 D-ATOP$_1$ 是 Log-APX 困难的，要证明 (d)，只需证明 D-ATOP$_1$ 属于 Log-APX。我们可以通过构造一个从 D-ATOP$_1$ 到 MSC 的 S-归约来证明这一点。由于我们知道 MSC 属于 Log-APX，这就意味着 D-ATOP$_1$也属于 Log-APX。

从 D-ATOP$_1$ 到 MSC 的 S-归约可以按如下方式构造。对于一个 D-ATOP$_1$ 实例 $(C, \{g\})$，我们定义一个 MSC 实例，并为每个输入 $i \in I$创建一个集合 S_i 。这些集合中的元素是原 D-ATOP$_1$ 问题中的错误函数。此外，每个集合 S_i 包含那些对输入 i产生错误输出的函数。也就是说：

$$S_i = \{f : f \in C \backslash \{g\}, f(i) \neq g(i)\} \quad \forall i \in I$$

那么，集合 $\{S_{l_i} \colon i \in \{1, \ldots, q\}\}$ 是一个集合覆盖，当且仅当每个函数 $f \in C\backslash\{g\}$ 都至少包含在一个集合 S_{l_i}中。该性质成立，当且仅当对每个函数 $f \in C\backslash\{g\}$，至少存在一个输入 l_i，使得 $f(l_i) \neq g(l_i)$；这意味着，按照任意顺序应用集合 $\{l_i \colon i \in \{1, \ldots, q\}\}$ 中的所有输入，都是原测试问题的有效测试策略。

值得注意的是，两种问题中的解的代价相同，q，即使 MSC 的解是最优的，结论也成立。因此，我们有一个从 D-ATOP$_1$ 到 MSC 的 S-归约。由此，D-ATOP$_1$ 属于 Log-APX。由于我们已经证明 D-ATOP$_1$是 Log-APX 困难的，因此可以得出 D-ATOP$_1$ 是 Log-APX 完全的。

此外，通过将该 S-归约用作多项式归约，我们还证明了 D-ATDP$_1$ 属于 NP，因为 MSC 属于 NP。由于我们先前已证明 D-ATDP$_1$ 是 NP-困难的，因此得出 D-ATDP$_1$ 是 NP 完全的，如 (b) 中所述。□

分析完确定性情形后，接下来我们讨论非确定性情形。

定理 2 我们有，N-ATDP 和 N-ATDP$_1$ 都是 PSPACE 完全的。 □

证明概要：完整证明见附录，这里我们介绍主要思路。首先，我们证明 N-ATDP 属于 PSPACE，因此，

我们也将证明其特例 N-ATDP$_1$ 也属于 PSPACE。然后，我们证明 N-ATDP$_1$ 是 PSPACE 困难的，因此，我们也将证明其推广形式 N-ATDP 也是 PSPACE 困难的。

N-ATDP属于 PSPACE

下面给出一种算法，它能够使用相对于输入规模呈多项式量级的内存来解决 N-ATDP。我们假设 I 中的元素存在一个全序， O中的元素也存在一个全序，并且我们可以使用相对于集合 I 和 O规模呈多项式量级的内存比较任意一对元素。这显然可以做到，因为我们可以使用 $|I|^2$ 和 $|O|^2$ 内存显式定义所有可能的元素对。

我们的算法将使用两个向量 i 和 o，以及一个自然数变量l。向量 i（分别是 o）将存放当前正在研究的候选测试策略分支中所考虑的输入（分别是输出）变量，而自然数变量 l 将存放我们在决策树中所处的深度。假设在N-ATDP实例中，测试策略最多可以应用于 k 个输入，那么显然， k 应当是向量 i 和 o的大小，同时也是 l可能的最大值。然而请注意，在 N-ATDP实例中，k 所表示的数值相对于表示它的二进制编码长度而言是指数级的。因此，为了保证内存使用量为多项式量级，向量 i 和 o 的大小必须小于k 。它们的实际大小将在后文说明。

该算法的伪代码见附录4，它以极小极大的方式尝试输入和输出的所有可能组合。如果某个组合既不能排除所有正确函数，也不能排除所有错误函数，那么这个组合就没有用。因此，我们修改最后一个尚未穷尽其所有可能值的输入，也就是尝试下一个输入可能性。如果没有更多输入可能性，就意味着判定问题的答案是否定的。

如果某个给定组合可以排除所有正确函数或所有错误函数，那么我们就修改最后一个尚未穷尽其所有可能值的输出，也就是继续检查新的输出是否会改变这一策略的可行性。如果没有更多输出可能性，就意味着我们已经能够在所有可能情况下排除所有正确函数或所有错误函数。也就是说，判定问题的答案是肯定的。

尽管所遍历的树规模呈指数级增长，但在算法执行期间，程序只需记住自己在树遍历过程中的位置，而向量 i 和 o 以及自然数 l 即可确定该位置。请注意，无论 N-ATDP 实例（C， E， k）中给出的 k的实际值是多少，为判定 N-ATDP 而构造的测试策略深度都不必超过 h，其中 $h = |I|$ 是可用输入的数量，因为在同一分支中重复输入毫无用处。请注意，N-ATDP 中函数的非确定性使得被测实现无论接受相同输入多少次，都可以对其给出相同输出。因此，如果测试策略中的某个节点应用了重复输入，即此前同一分支的某个节点已经应用过的输入，那么对于该节点的某个子节点，未被排除的

函数集合将与其父节点中的集合完全相同。因此，在前一个节点处排除所有正确函数或所有错误函数的问题，与在后一个节点处的问题相同——尽管后一个节点多应用了一次输入，而这次重复输入毫无用处。由此可知，算法执行的遍历深度永远不必超过 h，因此向量 i和 o的实际大小，以及 l可能达到的最大值，均可定义为 $min(k,\ h)$。由于在任何 N-ATDP 实例（$C,\ E,\ k$）中，可用输入数量 h 都不会超过 C 的表示大小，因此所遍历树的深度是多项式级的，向量 i 和 o 可以用多项式内存表示。

此外，算法的每个具体操作只需要常数个辅助变量。注意，我们只需取得给定集合中的最小值或最大值、取得集合中的下一个元素、检查对于所有 $1 \leq m \leq min(k,\ h)$，$o[m] \in f(i[m])$ 是否成立，以及检查 $f \in E$或 $f \in C \backslash E$ 是否成立。由此可知，该算法在多项式空间内运行。

N-ATDP$_1$ 是PSPACE 困难

在这种情况下，我们必须将一个 PSPACE 困难问题多项式归约到我们的问题 N-ATDP$_1$。具体而言，我们将全称量化布尔公式问题（TQBF [30, 31]）归约到 N-ATDP$_1$。在这个问题中，给定一个命题逻辑公式，并用全称量词和存在量词对其所有变量进行量化，我们需要判断所得的一阶逻辑公式是否为真，即是否等价于 $\top$。这个问题有多种表述方式；具体而言，我们假设命题公式以合取范式（CNF）给出，且存在量词和全称量词交替出现。也就是说，公式始终具有以下形式：

$$\exists x_1 \forall y_1 \exists x_2 \forall y_2 \ldots \exists x_k \forall y_k\ \phi$$

其中，ϕ是定义在变量 x_1、y_1、.....、x_k、y_k上的命题逻辑公式，该公式以 CNF 形式给出，即 $\phi = c_1 \wedge \ldots \wedge c_n$由若干析取子句$c_1$、.....、$c_n$组成。

如果将两个问题都视为博弈，我们就更容易介绍从全称量化布尔公式到 N-ATDP$_1$ 的多项式归约：在全称量化布尔公式中，“玩家”试图依次设置存在量化变量，使得无论“对手”如何设置相应的全称量化变量，公式都为真；在 N-ATDP$_1$中，测试者依次选择施加于被测实现的输入，使得无论“对手被测实现”如何选择相应的输出，最终都会舍弃所有正确函数或所有错误函数。我们的归约会将全称量化布尔公式中的存在量化变量映射为 N-ATDP$_1$ 的输入（由测试者选择），并将全称量化布尔公式中的全称量化变量映射为随后产生的相应 N-ATDP$_1$ 输出（由被测实现选择）。因此，当测试者引入某个输入，隐式地设定存在量化变量 x_j 的值时，被测实现将以输出作答，隐式地设定全称量化变量 y_j 的值。我们将看到，在某些变量取值下，公式 ϕ 为真，当且仅当在相应的输入和输出下，所有正确函数或所有错误函数均被舍弃。特别地，全称量化布尔公式中的每个析取子句 c_i ϕ都将由一个

N-ATDP$_1$ 中的错误函数 f_{c_i} （实际上，满足 c_i 等价于舍弃 f_{c_i}）表示，而 N-ATDP$_1$ 实例将只包含一个正确函数g。由于一旦唯一的正确函数被舍弃，测试者便会立即获胜，因此，在被测实现输出始终与该唯一正确函数一致的情况下，如何将被测实现在所有情况下分类为正确或错误，才是问题中困难且最有意思的部分。

更具体地说，给定一个形式为 $\exists x_1 \forall y_1 \ldots \exists x_k \forall y_k \phi$ 的 TQBF 实例，我们定义一个 N-ATDP$_1$ 实例，其中包含：对每个存在量化的变量设置两个不同的输入，一个表示 x_j 被设为 $\top$，另一个表示它被设为 $\bot$；一个正确函数 g，对所有输入都可以返回 0 或 1；三个可能的输出 0、1，以及 -1，分别表示变量 y_j 的两个可能值，以及一个新的额外值（-1），以便将任何产生该值的函数与无法产生该值的正确函数区分开来；对于逻辑公式 ϕ 的每个析取子句（c_i），还有一个错误函数（f_{c_i}）。形式化地说，

- 测试策略允许的深度，即 N-ATDP$_1$实例的第三个值，设为 k。

- $I = \{x_1, x_2, \ldots, x_k\} \cup \{\overline{x_1}, \overline{x_2}, \ldots, \overline{x_k}\}$,
- $O = \{0, 1, -1\}$,
- $E = \{g\}$，其中 $\forall i \in I \; g(i) = \{0, 1\}$,
- $C = E \cup \{f_{c_i} : \; c_i \in \phi\}$ 其中

$$f_{c_i}(x_j) = \begin{cases} -1 & \texttt{if } x_j \in c_i \\ -1 & \texttt{if } y_j, \overline{y_j} \in c_i \\ 0 & \texttt{if } y_j \in c_i \; \wedge \; x_j, \overline{y_j} \notin c_i \\ 1 & \texttt{if } \overline{y_j} \in c_i \; \wedge \; x_j, y_j \notin c_i \\ \{0,1\} & \texttt{otherwise} \end{cases}$$

$$f_{c_i}(\overline{x_j}) = \begin{cases} -1 & \texttt{if } \overline{x_j} \in c_i \\ -1 & \texttt{if } y_j, \overline{y_j} \in c_i \\ 0 & \texttt{if } y_j \in c_j \; \wedge \; \overline{x_j}, \overline{y_j} \notin c_i \\ 1 & \texttt{if } \overline{y_j} \in c_i \; \wedge \; \overline{x_j}, y_j \notin c_i \\ \{0,1\} & \texttt{otherwise} \end{cases}$$

当且仅当对应变量取值下，原始 TQBF 实例中的子句 c_i 满足时，每个函数 f_{c_i} 都会在 N-ATDP$_1$ 实例中因某些输入和输出而被舍弃。如果文字 x_j（x_j 同理）出现在析取子句 c_i 中，那么当 x_j 成立时，c_i 显然满足。此时，输入x_j （分别地,— x_j）可将 f_{c_i} 与正确函数 g 区分开来，因为 f_{c_i}对该输入返回 -1，而 g 返回 0 或 1。注意，如果被测实现返回-1，测试者就会立即通过舍弃唯一正确函数 g 来获胜；如果返回 0 或 1，则测试者会舍弃 f_{c_i}。此外，如果 y_j 和 y_j 都属于 c_i，那么 f_{c_i} 对输入 x_j 和 x_j 都会返回 -1，因为 c_i在这两种情况下都会显然满足。否则，如果文字 y_j （同理，对于

$\overline{y_j}$）出现在子句 c_i 中，那么 f_{c_i} 被定义为返回与满足该文字的值相反的值。例如，如果文字 y_j 出现在子句 c_i 中，且被测实现对 x_j 或 x_j 返回 0，即被测实现将 y_j 设为 $\perp$，那么 f_{c_i} 就会被舍弃，也就是说，子句 c_i 得到满足，因为在这种情况下，f_{c_i} 被定义为返回 1。

前面的构造仍不正确，因为我们还必须解决两个问题。一方面，尽管构造出的 N-ATDP$_1$ 实例只允许使用 k 输入，测试者仍可能使用 k 输入，从而表示一个不可能的逻辑赋值。例如，对于 $k = 3$，测试者可以使用 x_1、x_1 和 x_3，这样就给 x_1 赋予了两种可能的值，同时使 x_2 未定义。另一方面，我们必须保证用于选择输入（即测试）的顺序，与相应全称量化布尔公式中变量出现的顺序相同。注意，在 N-ATDP$_1$, 中，输入（即测试）可以按任意顺序使用，但在全称量化布尔公式中，变量必须按规定顺序引入，而且这个顺序确实会产生影响。例如，很容易看出 $\exists x_1 \forall y_1 \exists x_2 \forall y_2((x_2 \wedge y_1) \vee (x_2 \wedge y_1))$ 成立，但 $\exists x_2 \forall y_2 \exists x_1 \forall y_1((x_2 \wedge y_1) \vee (x_2 \wedge y_1))$ 不成立。

我们将在 N-ATDP$_1$ 实例中加入新的待丢弃错误函数，并确保在所有情况下都不可能将它们全部丢弃（更准确地说，在测试者不会因为丢弃唯一的正确函数而轻易获胜 g的所有情况下），除非测试者为每个存在量化变量恰好给出一个赋值（上述第一个问题），并且所有赋值都按照 TQBF 实例要求的顺序应用（第二个问题）。我们会将另一个错误函数 f_0以及针对所有 $1 \leq j \leq k-1$ 的新错误函数 f_j 和 f'_j 加入集合 C。根据它们的设计，要丢弃它们全部，就会迫使测试者至少应用一个代表每个存在量化变量的输入，无论其为正向还是负向。鉴于我们的 N-ATDP$_1$ 实例中最多只能应用 k 个输入，这就解决了先前的第一个问题。

为了解决第二个问题，即顺序问题，新的错误函数将定义如下。对于任何表示存在量化变量 x_j 的值的输入（同样，值为正或负都无关紧要），f_j 和 f'_j 将分别回复 0 和 1。根据此定义，如果被测实现回复 1，就会丢弃 f_j；如果回复 0，就会丢弃 f'_j。或者，在被测实现回复 -1 的任何分支中，g 都会被丢弃，测试者也会轻而易举地获胜，因此接下来我们只关注这种情况没有发生的所有分支。请注意，丢弃 f_j 还是丢弃 f'_j，并不由测试者控制，因为这将由被测实现决定。我们将赋予测试者控制权：当应用与变量 x_{j+1} 对应的输入时，可以决定丢弃 f_j 和 f'_j 中的哪一个。为此，我们将所有表示存在量化变量的输入复制为素 版本和非素版本，这样，对于每个 j，我们都会有版本 x_j、x_j、x'_j 和 $\underline{x_j}'$。当向被测实现应用与 x_{j+1} 对应的输入版本时，如果使用非素版本（即 x_{j+1} 或 $\overline{x_{j+1}}$），函数 f'_j 必然会被丢弃，因为 f'_j 对这些版本都会返回 -1；而如果使用素版本（即 x'_{j+1} 或 x_{j+1}'），函数 f_j 就会被丢弃，因为对于这些版本，f_j 会返回 -1。选择要应用的 x_{j+1} 版本，就能让测试者控制丢弃 f_j 和 f'_j 中的哪一个；但同样，x_{j+1} 的任何版本都会在没有 测试者控制的情况下丢弃 f_{j+1} 或 f'_{j+1}。因此，测试者将

不得不使用合适的 x_{j+2} 版本来丢弃尚未被丢弃的函数，依此类推。

我们可以看到，丢弃所有函数 $f_0, f_1, f'_1, \ldots, f_{k-1}, f'_{k-1}$ 会迫使测试者按照确切的顺序应用所选的变量版本 $x_1, \ldots, x_k$；而作为这一过程锚函数的 f_0，会被 x_1 的任一版本丢弃。如果在将变量 x_{j+1} 的某个版本应用于被测实现之前，尚未应用 x_j 的对应版本，那么应用前者时，我们无法知道 f_j 和 f'_j 中哪一个之后不会被后者丢弃，因此，由 x_{j+1} 决定要挽救哪个函数将只能盲目进行。因此，测试策略中必然存在一个分支：在 f_j 和 f'_j 之间，x_j 的应用丢弃了那个已经被丢弃的函数，使另一个函数未被丢弃，从而导致测试者失败。

通过为每个 j 选择某个输入 $\overline{x_j}, x_j, x'_j$ 或 $\overline{x_j}'$，测试者将同时在否定与非否定轴上作出决定，以丢弃所有 f_{c_i} 函数；并在带撇与不带撇轴上作出决定，以丢弃所有 f_j 和 f'_j 函数。请注意，如果测试策略中存在一个分支，其中 g 以及某个 f_{c_i}、某个 f_l 或 f'_l 未被丢弃，测试者不会获胜。因此，满足所有子句并且通过对变量作一致赋值来保持变量顺序，是在 N-ATDP$_1$ 实例中获胜的必要条件。

根据前述讨论，N-ATDP$_1$ 实例现定义如下。所得多项式归约的正确性证明见附录 B。

- $I = \{x_1, x_2, \ldots, x_k\} \cup \{\overline{x_1}, \overline{x_2}, \ldots, \overline{x_k}\}$
$\cup \{x'_1, x'_2, \ldots, x'_k\} \cup \{\overline{x_1}', \overline{x_2}', \ldots, \overline{x_k}'\}$,
- k、O和 E 均不作修改。同样，对于所有输入 $i \in I$（包括新增的输入），$g(i) = \{0, 1\}$，
- $C = E \cup \{f_{c_i} : \ c_i \in \phi\} \cup \{f_0\} \cup \{f_l, f'_l : \ \forall l \in \{1, \ldots, k-1\}\}$，其中 f_{c_i}的定义与之前相同，适用于所有 $\overline{x_j}$ 或 x_j；对于新输入 x'_j（类似地，$\overline{x_j}'$），它返回的值与 x_j（类似地，$\overline{x_j}$）返回的值相同，即

  $$f_{c_i}(x'_j) = f_{c_i}(x_j) \qquad f_{c_i}(\overline{x_j}') = f_{c_i}(\overline{x_j})$$

  关于新的函数 f_l，我们使用其中的 $2k-1$ 个：对每个存在量化变量 x_j（f_j 和 f'_j）使用两个函数，但第一个变量除外，我们只为它添加一个函数（f_0）。每个变量及其否定对应的函数定义方式相同，即：

  $$\begin{array}{ll} f_l(x_j) = f_l(\overline{x_j}) & f_l(x'_j) = f_l(\overline{x_j}') \\ f'_l(x_j) = f'_l(\overline{x_j}) & f'_l(x'_j) = f'_l(\overline{x_j}') \end{array}$$

  这些函数的具体定义如下，适用于 $1 \leq l < k$:

$$
\begin{array}{ll}
f_l(x_l) = \{0\} & f_l(x'_l) = \{0\} \\
f'_l(x_l) = \{1\} & f'_l(x'_l) = \{1\} \\
f_l(x_{l+1}) = \{0,1\} & f_l(x'_{l+1}) = \{-1\} \\
f'_l(x_{l+1}) = \{-1\} & f'_l(x'_{l+1}) = \{0,1\} \\
f_l(x_j) = f'_l(x'_j) = \{0,1\} & \forall j \notin \{l, l+1\}
\end{array}
$$

最后，我们还有一个额外的函数 f_0 来处理锚点情形：

$$
\begin{array}{l}
f_0(x_1) = f_0(\overline{x_1}) = f_0(x'_1) = f_0(\overline{x_1}') = \{-1\} \\
f_0(x_j) = f_0(\overline{x_j}) = f_0(x'_j) = f_0(\overline{x_j}') = \{0,1\} \\
\quad \text{for} \;\; 1 < j \leq k
\end{array}
$$

显然，相对于 TQBF 实例的大小，该变换可以在多项式时间内完成：设 n 为 ϕ 中子句的数量，则 N-ATDP$_1$ 实例中有 $2k+n$ 个函数、 $4k$ 个输入和 3 个输出。我们在论文附录中证明，该变换确实将 TQBF 归约为 N-ATDP$_1$ 。

4 结论与未来工作

本文研究了这样一种情形下自适应测试的计算复杂性：IUT 的可能定义集合，以及每种定义针对每种可能交互的行为，都是按具体情况外延定义的。本文研究了允许或禁止非确定性的不同问题变体，也研究了可能正确的 IUT 定义集合或错误的 IUT 定义集合必须为单元素集或不必为单元素集的情形。由于复杂性困难性会通过推广而传递，我们的复杂性困难性结果不仅适用于简单模型，也适用于任何推广了该模型、表达能力更强的自适应测试模型。特别是，两个非确定性问题变体的 PSPACE 完全性表明，要使自适应测试成为 PSPACE 困难问题，完全不需要状态，因为这两个变体都假设了完全无记忆的设置。

关于实际应用，我们要指出，在许多实际场景中，列出可能发生的错误是很常见的。虽然逐一列出所有错误时，这份清单可能相当长，但出于实际需要，通常也可以用相当简洁的方式描述它们。附录 A 中的示例很好地说明了这一点：被测实现可能出现的错误种类相对较少，测试者可以轻松地列举出来。然而，这一规范很容易转换成一份极其冗长的可能实现清单。因此，尽管测试者无需逐一列出这些实现，仍然很容易得到这样一份详尽的枚举。

我们未来工作的主要方向——也是目前正在开展的工作——是应用启发式的非最优方法，以启发式方式求解自适应测试问题中的随机实例和设计实例。我们正在开发一种极小极大实现，用于处理 PSPACE 完全变体，即非确定性的

变体；此外，我们还在开发一种基于遗传算法的实现，用于处理确定性的变体。初步观测结果表明，这些方法有望在合理的执行时间内提供合理的次优测试策略。

致谢： 本研究部分得到项目 PID2019-108528RB-C22 的资助，也得到马德里自治区项目 S2018/TCS-4339（BLOQUES-CM）的资助，该项目由欧盟 EIE 基金共同资助。作者谨向匿名审稿人致谢，感谢他们对本文先前版本提出的宝贵建议。

附录

附录 A. 第二个动机示例：如何处理更大的场景

在第 2 节所示的示例中，需要考虑的可能 IUT 定义（函数）数量相当少，因此很容易手动枚举。在更复杂、更贴近实际的测试场景中，通过以测试者认为可能的任何方式组合所有考虑在内的可能正确选项和可能错误，可能会产生大量 IUT 定义。需要注意的是，由于在我们的外延式设定中不存在强制要求使用的规范模型——例如有限状态机、定时自动机、C++ 程序等——测试者完全可以自由决定在每个场景中哪些错误可能同时发生，哪些错误不能同时发生。需要指出的是，IUT 可能定义的集合非常庞大，并不意味着测试者要花很长时间来定义 工作 。实际上，这些庞大的集合通常应当是由数量更易管理的因素爆炸式组合而成，这些因素以紧凑的方式描述了所有允许和禁止的行为。在这种情况下，可以编写一个简单的程序，通过穷举所有因素的各种可能组合，枚举出由此产生的所有函数，即可能的 IUT 定义。这样，测试者就不必承担枚举任务。接下来，我们将介绍一个涉及大量函数的示例。

假设我们要测试一台已组装完成的自动取款机（ATM）。请注意，在这种情况下，每项测试的定义并不只是与 ATM 键盘交互的方式，因为 ATM 的初始配置也会影响测试结果。这一配置包括 ATM 中各种面额钞票的可用情况，以及测试期间连接到 ATM 的所有用户账户在银行数据库中的状态。因此，执行 ATM 的每个测试用例，都要在现金柜台存钞处放入确切数量的各种面额钞票（10€、 20€、 50€ 和 100€），修改数据库以创建余额符合要求的虚假用户，通过键盘与 ATM 交互，取出钞票，然后再次重置系统。整个过程可能需要几分钟，因此无法执行大量测试用例。因此，我们的测试

策略必须经过设计，以便在与被测实现的总交互次数有限的情况下，尽可能多地获取有关其正确性或不正确性的信息。

在这种设定下，一个完整的测试用例，即我们设定中的输入，由以下数据定义：

A 测试开始前，自动取款机中各种面额纸币的数量。B 对于测试用例中涉及的每个用户 ID，测试开始前该用户账户中的金额。C 对于同一测试中连续进行的每笔取款，需提供以下信息：发起取款的用户 ID；请求的金额；以及在系统要求确认时将给出的“是／否”确认。

信息 (A) 可表示为一个 4 元组，每种纸币类型对应一个元素；(B) 可表示为一组二元组（ID 和账户中的金额）；(C) 可表示为一个有序元组列表。因此，一个测试用例可表示为包含这三项的三元组。例如，

$$\begin{aligned}&(\,(24,\ 13,\ 14,\ 23),\ \{(ID_1,\ 100),\ (ID_2,\ 140)\},\ [(\\&ID_1,\ 80,\ yes),\ (ID_2,\ 200,\ no),\ (ID_1,\ 40,\ yes)]\,)\end{aligned}$$

表示这样一种测试：首先向自动取款机提供 24 10€ 张纸币、 13 20€张纸币、 14 50€ 张纸币和 23 100€ 张纸币；接着创建虚假的用户账户 ID_1 和ID_2 ，其余额分别为 100€ 和 140€；最后与自动取款机交互，让这两位用户连续进行三次取款尝试： ID_1尝试取款 80€，接着 ID_2 尝试取款 200€，最后 ID_1 再次尝试取款 40€。

对于每次测试的输出，其中包含每笔取款请求的以下信息：实际支付给用户的钱款，具体来说，是支付的各类纸币数量；屏幕上显示的该用户余额信息，这可能不同于系统内部记录的实际余额，而后者用户并不知情；以及是否要求用户确认“是/否”。此外，我们假设能够检测到每当自动取款机试图支付一张纸币时，该纸币按其内部数据应当存在于纸币存放区，但实际上并不存在，因此，这类错误也会纳入与每笔取款相关的信息中。因此，每次测试的输出将表示为一个列表，其中每个元素都包含上述每笔取款的全部输出信息。

关于规范，即被认为正确的所有可能 IUT 定义的集合，实现者在若干方面拥有一定的自由度。我们考虑以下选项：

- （2 种选项）将每笔 100€金额都用一张 100€纸币支付；或者用两张面额各为 50€ 的纸币支付。如果其中一种纸币用完，则剩余金额改用另一种纸币支付。

- （2 种选项）将待支付金额除以 100 后的余数，先尽可能多地使用自动取款机中可用的 20€ 纸币支付，再用 10€ 纸币补足；或者先尽可能多地使用 10€ 纸币支付，如果 10€纸币用完，则用 20€ 纸币补足。

- (2 种选择) 如果自动取款机中存入的现金不足，被测实现可以给出自动取款机中剩余的全部现金，也可以什么都不给。

- (3 种选择) 如果客户的账户余额不足，被测实现可以恰好给出账户中剩余的金额、什么都不给，或者赊账。在后一种情况下，需要请求确认。

- (2 种选择) 对前两项中可能给出部分金额的所有情况请求确认；或者不请求确认。
- (2 种选择) 在给出所请求的现金或显示无法完成操作之前，展示银行投资基金的广告；或者展示银行贷款的广告。

对于前六种选择中的每一种，每个正确实现都可以确定性地遵循其中一种可用选项。此外，在最后一种选择中，还允许遵循一个额外选项：在每种情况下非确定性地选择两种选项中的任意一种。因此，这种选择产生了 3 种可能性，而不是 2 种。以任何可能的方式组合由此产生的选项后，我们的规范集合中就有 $2 * 2 * 2 * 3 * 2 * 3 = 144$ 个可能的正确函数，也就是可能的 IUT 定义。

关于可能的错误函数集合，我们考虑以下可能的错误：

a 在同一测试中进行第二次及之后的所有取款时，无论测试是否尝试使用不同的用户，都假定使用第一个用户 ID 的数据。 b 每次取款后，不更新自动取款机中可用纸币的数量。 c 每次取款后，不更新用户账户中的可用金额。 d （ 2 种选项）比应发放的最大面额纸币多发或少发一张。 e （ 2 种选项）忽略两项允许的标准，将金额中为 100 的倍数的部分以 100€ 和 50€ 的替代钞票发放，直至库存不足；或者完全忽略这部分金额，即不予发放。

f（2 种选项）忽略两项允许的标准，将金额的剩余部分（即不足 100 的部分）以 20€ 和 10€ 的替代钞票发放，直至库存不足；或者完全忽略这部分金额，即不予发放。

g（6 种选项）认为任何类型的纸币 $x \in \{10, 20, 50, 100\}$ 均可与类型为 $y \in \{10, 20, 50, 100\}$、带有 $x \neq y$ 的纸币互换，无论是用于发放还是用于统计自动取款机中每种纸币的可用数量。由于这会在类型为 x 的纸币与类型为 y、带有 $x \neq y$ 的纸币之间建立等价关系，且 x 与 y 的顺序无关，因此会产生 6 种选项。

h 不论用户在银行账户中的余额有多少，都发放其要求的全部金额。

i 从不显示任何广告。

j（2 种选项）在确认请求中，内部互换“是”和“否”的回答；或者认为所有回答都与第一次取款时给出的回答相同。

这为我们提供了 $1 + 1 + 1 + 2 + 2 + 2 + 6 + 1 + 1 + 2 = 19$ 种可能的错误。对于上述提到的 144 个正确规格，我们考虑多个包含上述一种或多种错误的错误版本。测试者假定开发团队犯下超过 5 个错误的概率可以忽略不计，因此，在这 144 个可能的正确 IUT 定义中，以任何方式引入 0 到 5 个错误共有 $\binom{19}{5} + \binom{19}{4} + \binom{19}{3} + \binom{19}{2} + \binom{19}{1} + 1 = 11,628 + 3,876 + 969 + 171 + 19 + 1 = 16,664$ 种可能。因此，IUT 共有 $144 * 16,664 = 2,399,616$ 种可能定义，其中 144 个是正确的。

由于执行每个测试用例需要几分钟的时间，我们希望尽量减少需要执行的测试数量；与预设测试相比，采用自适应测试可以减少这一数量。请注意，即使最初的测试没有表现出任何自身就可判定为故障的行为，即在检查其输出与其他测试结果的一致性之前，它们也能告诉我们接下来应执行哪些其他测试，从而更高效地追踪潜在故障。例如，

- 如果某个测试表明，金额中为 100 的倍数的部分是用 100€ 张钞票支付的，那么要检查 50€ 张钞票是否得到正确处理，就需要在后续测试中放入数量不足的 100€ 张钞票，否则自动取款机不会使用它们。如果金额的这一部分是用 50€ 张钞票支付的，则情况正好相反；
- 对于 100 的余数和 20€−10€ 张钞票，我们也有类似的情况；
- 如果某项测试检测到，当用户尝试取款金额超过自动取款机可用余额时，自动取款机会选择发放自动取款机中可用的全部现金

（而不是拒绝取款），那么要检查自动取款机中的钞票清点是否按预期工作，就需要更少的交互：在某些测试中，自动取款机会准确告诉我们它认为自己有多少张钞票；

- 如果某项测试检测到系统未请求确认，那么就无需执行旨在检测确认过程中的故障的测试。

这说明，测试用例的执行顺序通常会影响我们需要执行多少测试用例才能收集到有关被测实现正确性／错误性的有用信息：如果最先执行的测试用例侧重于根据这些测试收集到的输出，判断之后执行哪些其他测试用例会有用，那么通常只需较少的交互即可得出正确性／错误性的诊断。因此，测试策略中测试用例的顺序，尤其是各个分支中的顺序，至关重要。

如前所述，一个简单的程序可以通过以测试者允许的所有方式组合所有正确和错误的备选项，枚举被测实现的所有可能的正确和错误定义。一般来说，这种方法赋予测试者充分的灵活性，使其能够反映自己对被测实现开发方式的了解或怀疑。例如，故障 X 和 Y 可能不会同时发生，但故障 X 和 Z 可能会同时发生；如果开发人员选择了合法选项 W，则预计不会出现故障 X；等等。

在由此得到的枚举中，每个可能的被测实现定义都将是一个将输入（即测试用例）与输出关联起来的函数。实际上，只需为测试者认为重要且具有代表性的某个有限测试用例集定义函数的行为，随后便可据此设计合适的测试策略。这个有限测试用例集通常比我们能够对被测实现执行的测试用例数量更多，它将从所有可能的测试用例中提取出来；集合中的每个测试用例都可以选择纳入或不纳入测试策略。

一般来说，这些先验重要的测试用例，都会根据被测实现的性质，从若干有限、离散和/或连续的选择域中组合取值。这些可能性分别包括：简单的选择（如“是”/“否”）、要提取的欧元金额（如 70€或 450€），以及连续按键操作之间的时间间隔（如 3.47 秒或 0.1242 秒）。[5] 请注意，对于涉及连续域或无限离散域的选择，必须从无限多种可能性中选出一些代表性元素，纳入所考虑的测试用例中。应选取极端值和标准值：前者是非常小和非常大的值；后者则可通过选取若干值来获得，这些值的预期分布应与被测实现在实际使用中预期面对的分布相同。例如，如果某个实数的预期分布符合

[5]在我们的示例中，测试用例不考虑这些时间间隔，因为没有任何功能依赖于它们。如果加入一项功能，例如在超时时注销，那么测试用例就需要明确提及自动取款机中连续交互之间的时间间隔。

参数为 $\lambda = 3$的指数变量，那么可以在所考虑的测试用例中组合使用该变量的若干个样本，以及一些非常小和非常大的值，以观察退化行为。

请注意，我们无需为表示每个函数都存储一个程序：一个由合法选择和故障选择组成的简单向量即可表示每个函数，而一个接收这些选择的通用程序将告诉我们该向量所表示的相应函数的行为。

在我们的示例中，输出并不表示连续值，因为此处没有任何系统需求依赖于连续域。或者，我们可以考虑一些输出确实表示连续值的情景，那么就需要处理如下输出，例如： $o =$ “信号 Ok 在 3.448 秒后显示。”在这种情况下，在测试过程中匹配观测到的被测实现输出时，应允许一定的误差范围。因此，输出 o 应重新定义为例如“信号 Ok 在 3.448 ± 0.05 秒后显示”。请注意，这些区间可能重叠，从而在测试过程的观测层面产生非确定性。例如，设 $f \in E$和 $f' \in C \backslash E$，其中 $f(a) = \{1.4\}$，且 $f'(a) = \{1.5\}$。也就是说，根据 f所表示的被测实现的一种可能的正确的定义，数值输出 1.4 会对输入 a作出确定性响应；而根据 f'所表示的一种可能的错误的定义，输出 1.5 会对 a作出确定性响应。设测试者的观测精度范围为 ± 0.2。假设被测实现的实际定义为 f。请注意，收到输入 a时，被测实现的 1.4 响应在测试者看来会是 $[1.2, 1.6]$范围内的任意值。另一方面，如果被测实现的实际定义为 f'，那么 1.5 这一响应在测试者看来会是 $[1.3, 1.7]$范围内的任意值。请注意，子区间 $[1.3, 1.6]$ 是两个函数共有的，而区间 $[1.2, 1.3)$ 和 $(1.6, 1.7]$内的任意值，都能唯一确定产生该响应的函数分别是 f 或 f'。为纳入这一可观测性限制，输入 a时，f和 f'的行为重新定义如下： $f(a) = \{o_1, o_2\}$和 $f'(a) = \{o_2, o_3\}$，其中这三个新输出 o_1、o_2、 o_3 抽象地表示互不相交的数值区间 $[1.2, 1.3)$、 $[1.3, 1.6]$和 $(1.6, 1.7]$。

因此，尽管 f 和 f' 的原始定义分别对输入 a产生确定性的响应 1。4 和 1。5，但重新定义后，它们的响应变为非确定性的，以反映我们区分某些观测结果与实际输出的能力有限。

尽管本文的目标仅仅是展示如何根据先前的数据寻找测试策略的计算复杂性，接下来我们将简要概述在实践中如何通过启发式方法应对这些问题。测试策略可以动态构建，方式类似于在博弈过程中实时计算其他 PSPACE 完全博弈的策略。收集每个测试用例的结果后，便可排除一些可能正确或错误的 IUT 定义；随后应在有限深度内应用极小极大算法——这是因为搜索空间的规模呈指数增长——并将算法选出的最佳首步所对应的测试用例，作为下一步测试应用于被测实现。最有趣的分支可以探索得更深，而在该最大深度上的节点通常应采用某种启发式方法。例如，可以用可能的正确

定义与可能的错误定义之比，作为这些节点的启发式方法。

实际上，在复杂测试场景中，现实的目标不应是保证在任何情况下都能舍弃所有可能正确或所有可能错误的 IUT 定义。相反，目标应是达到这样的配置：可能正确的定义与可能错误的定义之间的比例极端地偏向任一方——也就是说，即使少数一方没有被完全舍弃，其相对概率也非常低。因此，实际目标并非实现穷尽性并给出精确的正确／错误诊断，而只是达到某种伪穷尽性。

考虑到执行极小极大算法来决定每个要应用于 IUT 的新测试用例需要耗费不可忽略的时间（具体取决于算法试图达到的深度），必须适当调整该深度，使算法执行时间相对较短——尤其是与将每个测试用例应用于 IUT 所需的时间相比。否则，测试过程就会因等待决定下一个应应用哪个测试用例而中断一段不可忽略的时间，从而浪费大量时间。请注意，在这种情况下，与其执行算法并等待结果，不如将这段时间用于应用大量任意测试用例：即使这些测试用例并非依据任何算法驱动的标准来选择（例如随机选择），应用更多测试用例也可能胜过应用少量经过精心设计的测试用例。还要指出，算法的执行与向 IUT 应用额外测试可以轻而易举地并行进行，因此可以轻松兼顾两种选择的优点：我们既可以应用算法选出的那些巧妙的测试用例，也可以应用其他随机设计的测试用例，从而无需让测试过程停下来等待极小极大算法决定下一个（合适的）待应用测试用例。此外，极小极大算法的执行不仅能为我们提供下一个最适合应用的测试用例，还能提供其他值得考虑的后续测试用例——即那些能够到达极大极小值较高的节点的其他走法——因此，至少其中一部分待应用的其他测试用例或许也并非完全随机。

当预计每个测试用例的执行都需要几分钟时，就不需要将最佳测试用例（最佳的第一步）与其他测试结合起来使用，前文给出的示例便是如此。在这种情况下，极小极大算法运行到适当深度所需的时间，远短于对被测实现应用任何测试用例所需的时间，因此无需用前面提到的那些测试用例来填补“测试空闲时间”：对被测实现应用的所有测试，恰好都是极小极大算法找出的最佳走法。

附录 B. 定理 2 的证明

B.1 在多项式空间中求解 N-ATDP 的算法

我们在前文非正式描述的、以多项式空间求解 N-ATDP 的算法如下：

$l = 1$; $h = |I|$; $k = min(k,\ h)$[步骤 0]
$i[l] = \min(I)$[步骤 1]$o[l] = \min($
$\bigcup_{\forall m<l} f(i[l]))_{f\in\{f\in C:\ o[m]\in f(i[m])}$[步骤 2]

- 如果 $l < k$:– $l = l + 1$; 转到
[步骤 0]。

- 如果 $l = k$: – 如果 $\{f \in C :\ o[m] \in f(i[m])\ \forall m \le k\}$ 是 $C\backslash E$ 或 E的子集，也就是说，不存在这样的 $f \in E$ 使得对于所有 $1 \le m \le k$ 都有 $o[m] \in f(i[m])$，或者不存在这样的 $f \in C\backslash E$ 使得对于所有 $1 \le m \le k$都有 $o[m] \in f(i[m])$：当（$l > 0$ 且$o[l] = \max \bigcup_{o[m]\in f(i[m])\ \forall m<l} f(i[l])$）时：$l = l - 1$如果 $l = 0$: 返回 YES。否则，将 $o[l]$赋给下一个输出，并转到 [步骤 2]。– 如果 $\{f \in C :\ o[m] \in f(i[m])\ \forall m \le k\}$ 既不是 $C\backslash E$ 的子集，也不是 E的子集：当（$l > 0$ 且$i[l]$ 是满足 $a \ne i[m]\ \forall\ 1 \le m \le l - 1$的最大输入时）$l = l - 1$。如果 $l = 0$: 返回 NO。否则，将 $i[l]$ 赋给下一个输入 $a \in I$，满足 $a \ne i[m]\ \forall\ 1 \le m \le l - 1$，并转到 [步骤 1]。

显然，为了解决优化问题而非判定问题，我们可以依次尝试 $k = 1$、 $k = 2$ 等，直到找到第一个返回 YES 的值。

前述算法的效率还可以进一步提高。例如，该算法仅在层级 $min（k,\ h）$ 检查所有函数是否都属于 E 或是否都属于 $C\backslash E$，但这种情况也可能更早出现，而且在其出现后继续扩展当前分支是不必要的。尽管如此，该算法本身仍可在多项式内存内运行。特别是，正如我们前面所指出的， i、 o 和 l 通过树定义了探索点，并且它们可以用多项式内存来定义。

B.2 N-ATDP$_1$ ∈PSPACE 困难

根据定理 2 证明概要中给出的归约，我们必须证明：任意一个全称量化布尔公式的实例成立，当且仅当相应的 N-ATDP$_1$ 实例成立。

B.2.1 TQBF $\Rightarrow$ N-ATDP$_1$ 我们首先证明，如果 TQBF 公式成立，那么对应的 N-ATDP$_1$ 实例也成立。

首先，我们证明，如果按照正确的顺序恰当地选择与存在量化变量对应的输入，那么要么 g，要么所有 f_l（和 f_l'）函数都会被丢弃，具体如下。第一个输入可以是任意值$i_1 \in \{x_1, x_1, x_1', x_1'\}$。之后，对于其余输入，我们必须根据上一次得到的输出，从集合$\{x_l, x_l\}$或集合 $\{x_l', x_l'\}$中恰当地选择，以丢弃 f_l 或 f_l'。形式化地，对于每个 $1 \leq l < k$:

- 如果 $o_l = -1$，那么正确函数 g 会被丢弃，我们便能确定被测实现是错误的。因此，在这个策略分支中不再需要更多输入；
- 如果 $o_l = 0$（不丢弃 f_l），那么我们使用 $i_{l+1} \in \{x_{l+1}', x_{l+1}'\}$（会丢弃）；
- 如果 $o_l = 1$（不丢弃 f_l'），那么我们使用 $i_{l+1} \in \{x_{l+1}, x_{l+1}\}$（会丢弃）。

因此，如果任何输出为 -1，那么所有与输入和输出一致的函数都是错误的，因为唯一的正确函数g 从不返回 -1，即丢弃了 g。否则，所有输出都属于集合 $\{0, 1\}$，但所有函数 f_l和 f_l'都会被丢弃。注意，对于所有$1 \leq l < k$，我们针对 o_l有两种情况：

$$o_l = 0 \Rightarrow \begin{cases} o_l \notin f_l'(i_l) = \{1\} \\ i_{l+1} \in \{x_{l+1}', \overline{x_{l+1}}'\} \Rightarrow \\ \quad 0 \leq o_{l+1} \notin f_l(i_{l+1}) = \{-1\} \end{cases}$$

$$o_l = 1 \Rightarrow \begin{cases} o_l \notin f_l(i_l) = \{0\} \\ i_{l+1} \in \{x_{l+1}, \overline{x_{l+1}}\} \Rightarrow \\ \quad 0 \leq o_{l+1} \notin f_l'(i_{l+1}) = \{-1\} \end{cases}$$

因此，无论如何，对于至少一个输入，我们都会丢弃两者 f_l 和 f_l'，该输入可以是 i_l 或 i_{l+1}。最后，我们还必须丢弃函数 f_0。然而，考虑到 $i_1 \in \{x_1, x_1, x_1', x_1'\}$，我们知道$0 \leq o_1 \notin f_0(i_1) = \{-1\}$，因此会丢弃 f_0。

因此，我们得出结论：对于所有 $1 \leq l < k$，都有 f_l，f_l'，$f_0 \notin \{f \in C\colon\ o_m \in f(i_m)$ $m \leq k\}$。

现在我们证明，如果该合格布尔公式为真，那么要么观察到某个输出 -1——因此 g 被舍弃，且被测实现是错误的——，

要么所有函数 f_{c_i} 也都会被舍弃。如果在应用于 TQBF 实例的赋值策略中将变量 x_1 设为真，那么我们使用$i_1 \in \{x_1, x_1'\}$。否则，我们使用 $i_1 \in \{x_1, x_1{}'\}$。然后，对于其余的输入，也就是说，对于所有$1 \leq l < k$，我们有两种选择：如果 $o_l < 0$，那么不再需要其他输入，因为正确函数 g 会被 o_l舍弃，这是因为 g 永远不会返回 -1；否则，我们取 $y_l = o_l$，对于下一个输入又有两种情况：

$$\begin{cases} i_{l+1} \in \{x_{l+1}, x_{l+1}'\} & \text{if } x_{l+1} \text{ is true in the TQBF valuation} \\ i_{l+1} \in \{\overline{x_{l+1}}, \overline{x_{l+1}}'\} & \text{if } x_{l+1} \text{ is false in the TQBF valuation} \end{cases}$$

也就是说，要么所有输出都是正的，要么某个输出为负，此时我们就舍弃正确函数，因此所有与输入和输出一致的函数都是错误的。下面来看后一种情况下，所有形如 f_{c_i} 的函数都会被舍弃。

假设没有观察到负输出。那么，对于逻辑公式 ϕ 的所有析取子句 c_i 以及所有 $1 \leq j \leq k$,

- 若 x_j 为真，则 $i_j \in \{x_j, x_j'\}$。若 $x_j \in c_i$，则 $0 \leq o_j \notin f_{c_i}(i_j) = \{-1\}$。
- 若 x_j 为假，则 $i_j \in \{\overline{x_j}, \overline{x_j}'\}$。若 $\overline{x_j} \in c_i$，则 $0 \leq o_j \notin f_{c_i}(i_j) = \{-1\}$。
- 若 $y_j, \overline{y_j} \in c_i$，则 $i_j \in \{x_j, \overline{x_j}, x_j', \overline{x_j}'\}$. 因此，$0 \leq o_j \notin f_{c_i}(i_j) = \{-1\}$。
- 如果 $y_j \in c_i$ 但 $\overline{y_j} \notin c_i$ 且 y_j 为真，那么 $f_{c_i}(i_j)$ 要么是 $\{0\}$ 要么是 $\{-1\}$ 对于$\overline{i_j} \in \{\overline{x_j}, x_j, x_j', x_j'\}$且 $1 = y_j = o_j \notin f_{c_i}(i_j) \subset \{-1, 0\}$。
- 如果 $y_j \in c_i$ 但 $\overline{y_j} \notin c_i$ 且 y_j 为假，那么 $f_{c_i}(i_j)$ 要么是 $\{1\}$ 要么是 $\{-1\}$ 对于$\overline{i_j} \in \{\overline{x_j}, x_j, x_j', x_j'\}$且$0 = y_j = o_j \notin f_{c_i}(i_l) \subset \{-1, 1\}$。

总而言之，如果满足 c_i，即它包含一个值为真的文字，那么$f_{c_i} \notin \{f \in C : \ o_m \in f(i_m) \ m \leq k\}$。因此，如果满足 ϕ，那么它的所有子句都得到满足，并且 f_{c_i} 中没有任何一个属于 $\{f \in C : \ o_m \in f(i_m) \ m \leq k\}$。

需要指出的是，选择用于舍弃函数的输入 f_{c_i}，与选择用于舍弃函数 f_l、f_l' 和 f_0 的输入是兼容的。第二种情况下，我们在带撇号和不带撇号的输入之间进行选择；第一种情况下，我们则在正输入和负输入之间进行选择。综合考虑这两个因素后，除第一个输入外，在每种情况下都只有一个可用输入；对于第一个输入，则可使用带撇号和不带撇号的版本。因此，结合这两种策略，如果我们按适当的顺序引入变量且满足 ϕ，那么我们将舍弃 g 或所有其他函数，即形式为 f_{c_i}, f_l, f_0 的函数，随后解决 N-ATDP$_1$ 问题。

B.2.2 N-ATDP$_1$ ⇒ 全称量化布尔公式：现在我们证明，如果相应的 N-ATDP$_1$ 实例成立，那么合格布尔公式也成立。

首先，我们证明，若 (C, E, k) 满足 N-ATDP$_1$，那么在策略的每个分支上，要么存在某个输出 -1（并且 g 会被丢弃），要么对于所有 $1 \leq j \leq k$

我们至少有一个来自集合 $\{x_j, x_j, x_j', x_j'\}$ 的输入。因此，由于在任何分支上我们只能使用 k 个输入，对于后一类分支中的每个 x_j，也就是那些没有 -1 的分支，都恰好有一种可能的赋值。

假设（C， E， k）满足 N-ATDP$_1$，并假设同一分支上的输入和输出分别为 $\{i_m\}_{m=1}^k$ 和 $\{o_m\}_{m=1}^k$。如果存在 $o_m = -1$，那么该性质成立。因此，我们需要在另一种情况下寻找矛盾。也就是说，现在考虑这样的分支：对于所有 $1 \le m \le k$， $o_m \in \{0,1\}$。那么， $g \in \{f \in C\colon\ o_m \in f(i_m)\ i \le k\}$。此外，由于（$C$， E， k）满足 N-ATDP$_1$，我们有 $E = \{g\} = \{f \in C\colon\ o_m \in f(i_m)\ i \le k\}$；否则，测试将无法按满足 N-ATDP$_1$ 的要求区分错误函数与正确函数。下面证明，如果 $\exists j\colon\ x_j$， x_j， x_j'， $x_j' \notin \{i_m\}_{m=1}^k$，则该性质不可能成立：

- 如果 $j = 1$，那么无论使用什么其他输入，都无法将 f_0 与 g 区分开来，因为对于除对应于 $j = 1$ 的输入之外的所有输入， f_0 都等于 g。因此，$f_0 \in \{f \in C\colon\ o_m \in f(i_m)\ i \le k\}$，这就产生了矛盾，因为错误函数 f_0 将无法与正确函数 g 区分开来。

- 如果 $j > 1$ 且 $\nexists i_M \in \{x_{j-1}, \overline{x_{j-1}}, x_{j-1}', \overline{x_{j-1}'}\}$，那么就有两个连续的变量未被覆盖（下标分别为 j 和 $j-1$），但函数 f_l 与 g 仅在变量 l 和 $l+1$ 上不同。也就是说，对于所有 m，都有： $f_{j-1}(i_m) = f'_{j-1}(i_m) = g(i_m) = \{0,1\}$。因此$f_{j-1}$， $f'_{j-1} \in \{f \in C\colon\ o_m \in f(i_m)\ i \le k\}$，这又产生了矛盾。

- 如果 $j > 1$且 $\exists! i_M \in \{x_{j-1}, \overline{x_{j-1}}, x_{j-1}', \overline{x_{j-1}'}\}$，那么 o_M可以是 0 或 1。第一种情况下， $f_{j-1} \in \{f \in C\colon\ o_m \in f(i_m)\ i \le k\}$，因为 o_M不会丢弃 f_{j-1}，而且只有使用集合 $\{x_{j-1}, x_{j-1}, x'_{j-1}, x_{j-1}', x_j, x_j, x'_j, x_j'\}$中的输入才能丢弃此函数，我们没有更多此类输入。类似地，当 o_M 为 1 时，f'_{j-1}也会面临相同情况。无论哪种情况，都会产生矛盾，因为 f_{j-1}或 f'_{j-1}无法与 g区分。

- 如果 $j > 1$，且有 $\exists i_{M_1}$， i_{M_2}， ……， $i_{M_p} \in \{x_{j-1}, x_{j-1}, \overline{x'_{j-1}}, x_{j-1}', \}$，那么我们正在应用若干个与变量 x_{j-1}对应的输入。然而，对于 f_{j-1} （分别对于 f'_{j-1}），它们的返回值都将是 0（分别为 1），因此我们会处于与上一项相同的情况。也就是说， f_{j-1} 或 f'_{j-1} 将无法被丢弃。

现在证明，输入的引入顺序必然与 TQBF 实例定义中的顺序相同。也就是说，若 (C， E， k) 满足 N-ATDP$_1$，则 $\forall j\colon\ i_j \in \{x_j, x_j, x_j', x_j'\}$。

假设 (C， E， k) 满足 N-ATDP$_1$，并考虑一个策略分支，其输入集合为 $\{i_m\}_{m=1}^k$，相应的输出为 $\{o_m\}_{m=1}^k$。由于 (C， E， k) 满足 N-ATDP$_1$，，与这些输入和输出一致的 C 函数要么全部正确，要么全部错误。

如果存在一个值为 -1 的输出，那么显然有$\{f \in C\colon o_m \in f(i_m)\ i \le k\} \subseteq C \backslash E$，因为 $E = \{g\}$ 和 g只会返回

0 或 1。另一种情况（也就是说，对于所有 $1 \le m \le k$， $o_m \in \{0,1\}$ ），我们有$g \in \{f \in C\colon o_m \in f(i_m)\ i \le k\}$。因此，由于 $(C,\ E,\ k)$ 满足 N-ATDP$_{1,}$，我们必须有 $E = \{g\} = \{f \in C\colon o_m \in f(i_m)\ i \le k\}$。也就是说，我们丢弃其余的函数（所有错误的函数）。下面证明，如果输入的引入顺序不正确，就会产生矛盾。

考虑满足 $i_j \notin \{x_j, x_j, x'_j, x_j{}'\}$ 的最小值 j。那么，对于某个 $n > j$，我们有 $i_j \in \{x_n, x_n, x'_n, x_n{}'\}$。

我们将 $\{i_m\}_{m=1}^{k}$ 中属于 $\{x_{n-1}, x_{n-1}, x'_{n-1}, x_{n-1}{}'\}$ 的唯一输入记为 i_l。于是我们有 $l > j$。如果 $i_j \in \{x_n, x_n\}$ （分别为 $i_j \in \{x'_n, x_n{}'\}$），在 $o_j \in \{0,1\}$ 的情况下，我们将有 f_{n-1}（分别为 f'_{n-1}）$\in \{f \in C\colon\ o_m \in f\ (i_m)\ \ i \le \boldsymbol{j}\}$；而在策略中 $o_l = 0$（分别为 $o_l = 1$）的分支上，我们将有 f_{n-1}（分别为 f'_{n-1}）$\in \{f \in C\colon\ o_m \in f\ (i_m)\ \ i \le \boldsymbol{k}\}$，因为区分这些函数与函数 g 的唯一方法，是使用带有下标 n 或 $n-1$ 的变量。这就产生了矛盾。

既然我们知道每个变量都对应一个输入，且输入的顺序必然与变量的顺序相同，也就是说，每个输入都满足 $i_j \in \{x_j, x_j, x'_j, x_j{}'\}$，我们现在证明：只要 $(C,\ E,\ k)$ 满足 N-ATDP$_{1,}$，相应的公式 $\exists x_1 \forall y_1 \ldots \exists x_k \forall y_k\ \phi$ 就为真，即满足全称量化布尔公式。下面说明，按照从用于实例 $(C,\ E,\ k)$ 的 N-ATDP$_1$策略导出的全称量化布尔公式策略，每个子句 $c_i \in \phi$ 都得到满足。为了满足 ϕ，在每个全称量化布尔公式策略分支中，如果相应的 N-ATDP$_1$ 策略分支中 $i_j \in \{x_j, x'_j\}$，我们就将 x_j 设为真；如果$i_j \in \{x_j, x_j{}'\}$，则将其设为假。

提醒一下，每个 i_j （对于 $j > 1$）的值都取决于先前的输出 o_{j-1}。在我们的情形中，我们使用 $o_{j-1} = y_{j-1}$。注意，N-ATDP$_1$ 策略中带有某个 -1 输出的分支，会直接排除正确函数，并使测试策略在 TQBF 策略对应项中不存在的分支上获胜；而在 TQBF 策略中，变量只能为真或假。由于 TQBF 策略无法包含这类分支，接下来我们将关注其他分支。对于 N-ATDP$_1$ 策略中所有不存在 o_m 的分支，我们知道，对所有 $1 \le m \le k$，都有$o_m \in \{0,1\}$。因此，$g \in \{f \in C\colon\ o_m \in f\ (i_m)\ \ i \le k\}$。所以，由于我们知道（$C,\ E,\ k$）满足 N-ATDP$_{1,}$，因此在这些分支中有 $E = \{g\} = \{f \in C\colon\ o_m \in f\ (i_m)\ \ i \le k\}$。

在前述条件下，我们来证明：如果函数 f_{c_i} 被舍弃，即有 $f_{c_i} \notin \{f \in C\colon\ o_m \in f(i_m)\ i \le k\}$，那么相应的子句 c_i 得到满足。为了舍弃 f_{c_i}，必须存在一个输入 i_M，使得 $o_M \notin f_{c_i}(i_M)$。显然，如果 $f_{c_i}(i_M) = \{-1\}$，那么该函数就会被舍弃，因为我们考虑的是未观测到输出 -1的分支。根据构造，对于输入 i_M，函数 f_{c_i} 可以在以下三种情况下定义：

- $x_M \in c_i$ 且 $i_M \in \{x_M, x'_M\}$，因此 x_M 为真。
- $\overline{x_M} \in c_i$ 且 $i_M \in \{x_{\overline{M}}, x_{\overline{M}}{}'\}$，因此 x_M 为假。
- $y_M, \overline{y_M} \in c_i$。

显然，在这三种情况下，子句 c_i 均得到满足。

其余会丢弃 f_{c_i} 的情况如下：

- 若 $y_M \in c_i$，但 $x_M, \overline{y_M} \notin c_i$ 则 $f_{c_i}(i_M)$ 对于 $i_M \in \{x_M, x'_M\}$ 为 $= \{0\}$。
  因此，若丢弃 f_{c_i}，则 $o_M = 1$。
- 若 $y_M \in c_i$，但 $\overline{x_M}, \overline{y_M} \notin c_i$ 则 $f_{c_i}(i_M)$ 对于 $i_M \in \{x_M, x_M'\}$ 为 $= \{0\}$。
  因此，若丢弃 f_{c_i}，则 $o_M = 1$。
- 若 $\overline{y_M} \in c_i$，但 x_M，$y_M \notin c_i$ 则 $f_{c_i}(i_M)$ 对于 $i_M \in \{x_M, x'_M\}$ 为 $= \{1\}$。
  因此，若丢弃 f_{c_i}，则 $o_M = 0$。
- 若 $\overline{y_M} \in c_i$，但 $\overline{x_M}$，$y_M \notin c_i$ 则 $f_{c_i}(i_M)$ 对于 $i_M \in \{x_M, x_M'\}$ 为 $= \{1\}$。
  因此，若丢弃 f_{c_i}，则 $o_M = 0$。

如所见，如果舍弃了f_{c_i}，那么要么是 $y_M \in c_i$ 和 $o_M = y_M = 1$，要么是$y_M \in c_i$ 和 $\overline{o_M} = y_M = 0$。在这两种情况下，c_i 均得到满足。

在任何其他情况下，f_{c_i} 都不会被舍弃，因为在任何其他情况下，f_{c_i}（i_M）$=\{0, 1\}$。

正如我们已经看到的，每个 c_i 都得到满足，因此 ϕ也得到满足。

参考文献

[1] D. Lee 和 M. Yannakakis，《有限状态机测试的原理与方法：综述》，《*IEEE*汇刊》，第84卷，第8期，第1090–1123页，1996年。

[2] A. Petrenko，《基于故障模型从有限状态模型推导测试：注释书目》，载于第*4*届并行过程建模与验证暑期学校，*MOVEP'00*，*LNCS 2067*。Springer，2001年，第196–205页。

[3] R. Dorofeeva、K. El-Fakih、S. Maag、A. Cavalli 和 N. Yevtushenko，"基于有限状态机的一致性测试方法：附实验评估的综述"，《信息与软件技术》，第 52 卷，第 12 期，第 1286–1297 页，2010 年。

[4] J. Tretmans，"基于标记迁移系统的一致性测试：实现关系与测试生成"，Computer Networks and ISDN Systems，第 29 卷，第 49–79 页，1996年。

[5] ——，"并发系统测试：一种形式化方法"，载于第十届并发理论国际会议，*CONCUR'99*，*LNCS 1664*。Springer，1999年，第 46–65 页。

[6] E. Brinksma 和 J. Tretmans，"迁移系统测试：一份注释书目"，载于并行过程建模与验证第 *4* 届暑期学校，*MOVEP'00*，*LNCS 2067*。Springer，2001 年，第 187–195 页。

[7]J. Springintveld、F. Vaandrager 和 P. D’Argenio，“测试定时自动机”，载于理论计算机科学，第 254 卷，第 1–2 期，第 225–257 页，2001 年；此前作为特温特大学技术报告 CTIT-97-17 于 1997 年发表。[8] M. Krichen 和 S. Tripakis，“实时系统的黑盒一致性测试”，载于软件模型检验第 *11* 届国际 *SPIN* 研讨会，*SPIN’04*，*LNCS 2989*。Springer，2004 年，第 109–126 页。[9] I. Berrada、R. Castanet、P. Félix 和 A. Salah，“使用定时有界轨迹进行实时系统的测试用例最小化”，载于第 *18* 届 *IFIP TC6/WG6.1* 国际会议，*TestCom 2006*，*LNCS 3964*。Springer，2006 年，第 289–305 页。[10] M. Merayo、M. Núñez 和 I. Rodríguez，“扩展扩展有限状态机以规范和测试具有动作持续时间及超时的定时系统”，*IEEE* 计算机学报，第 57 卷，第 6 期，第 835–844 页，2008 年。[11] M. Stoelinga 和 F. Vaandrager，“概率自动机的一个测试场景”，载于自动机、语言与编程第 *30* 届国际学术会议，*ICALP’03*，*LNCS 2719*。Springer，2003 年，第 464–477 页。[12] N. López、M. Núñez 和 I. Rodríguez，“符号概率系统的规范、测试与实现关系”，理论计算机科学，第 353 卷，第 1–3 期，第 228–248 页，2006 年。[13] M. Efatmaneshnik、S. Shoval 和 K. Joiner，“系统测试架构评估：一种概率建模方法”，*IEEE* 系统杂志，第 1–12 页，2019 年。[14] L. J. Morell，“基于故障的测试理论”，*IEEE* 软件工程汇刊，第 16 卷，第 8 期，第 844–857 页，1990 年。[15] R. Hierons，“存在测试假设和故障域时测试集与标准的比较”，*ACM* 软件工程与方法学汇刊，第 11 卷，第 4 期，第 427–448 页，2002 年。[16] ——，“具有故障域或测试假设的测试中的判定函数”，*ACM* 软件工程与方法学汇刊，第 18 卷，第 4 期，2009 年。[17] I. Rodríguez、M. Merayo 和 M. Núñez，“$\mathcal{HOTL}$：假设与观测测试逻辑”，逻辑与代数编程杂志，第 74 卷，第 2 期，第 57–93 页，2008 年。[18]I. Rodríguez、L. Llana 和 P. Rabanal，“一般可测试性理论：类别、性质、复杂度与测试归约”，*IEEE* 软件工程汇刊，第 40 卷，第 862–894 页，2014 年。

[19] I. Rodríguez、F. Rosa 和 F. Rubio，“将复杂度引入形式化测试”，《逻辑与代数编程方法期刊》，第 111 卷，2020 年。[20] N. Kushik 和 N. Yevtushenko，“自适应归巢属于 P”，载于第十届基于模型的测试研讨会（*MBT 2015*），*EPTCS 180*，2015 年 4 月，第 73–78 页。[21] H. Yenigün、N. Yevtushenko 和 N. Kushik，“确定性有限状态机中检查自适应同步实验的存在性及其推导的复杂度”，《信息处理快报》，第 127 卷，第 49–53 页，2017 年。[22] N. Kushik、N. Yevtushenko 和 H. Yenigün，“降低检查非确定性有限状态机中自适应同步实验的存在性及其推导的复杂度”，载于面向验证与确认的特定领域模型方法国际研讨会，*AMARETTO@MODELSWARD 2016*。SciTePress，2016 年，第 83–90 页。[23] A. Petrenko 和 N. Yevtushenko，“对由非确定性有限状态机规约的确定性实现进行自适应测试”，载于软件与系统测试——第 *23* 届 *IFIP WG 6.1* 国际会议，*ICTSS 2011*，2011 年，第 162–178 页。[24] ——，“使用有限状态机对非确定性系统进行自适应测试”，载于第 *15* 届 *IEEE* 高可信系统工程国际研讨会，*HASE 2014*，2014 年，第 224–228 页。[25] P. van den Bos 和 F. W. Vaandrager，“带有输入和输出的标记迁移系统的状态识别”，*CoRR*，第 abs/1907.11034 卷，2019 年。 [在线]。可用地址：http://arxiv.org/abs/1907.11034[26] R. Bloem、G. Fey、F. Greif、R. Könighofer、I. Pill、H. Riener 和 F. Röck，“从时序逻辑规约综合自适应测试策略”，《系统设计形式化方法》，第 55 卷，第 103–135 页，2019 年。[27] I. Rodríguez，“一般可测试性理论”，载于 *CONCUR 2009*——并发理论，第 *20* 届国际会议，*LNCS 5710*。Springer，2009 年，第 572–586 页。[28] P. Crescenzi，“近似保持归约简明指南”，载于《计算复杂性论文集》——第十二届 *IEEE* 年度会议。IEEE，1997 年，第 262–273 页。[29] U. Feige，“用于近似集合覆盖的 ln n 阈值”，《*ACM*期刊》（*JACM*），第 45 卷，第 4 期，第 634–652 页，1998 年。[30] M. Sipser，《计算理论导论》。圣智学习，2012 年。

[31] L. Stockmeyer, 《问题的计算复杂性分类》, 符号逻辑杂志, 第52卷第1期, 第1–43页, 1987年。