A Testing Case Chain Based Method for the Failure Reproduction of Flight Control Software

Xiaoting Du, Nan Wang

School of Automation Science and Electrical Engineering, Beihang University, Beijing 100191, China
(xiaoting_2015@buaa.edu.cn)

***Abstract*—The reproduction of software failure plays an important role during the development and maintenance of software, especially makes a great contribution to the software debugging. Since the requirement of the highly real time performance and complex control algorithm, repetition of debugging is needed by the development and maintenance of flight control software. Consequently, the reproduction of failure is crucial to flight control software. Although many researches focus on the methodology of the reproduction of general software in recent years, it should be noted that these methods may be incompatible to flight control software whose characteristic is special comparing with general software. Therefore, it is necessary to explore the conformable method for the reproduction of the failure of flight control software. According to the characteristics of flight control software, a testing case chain was abstracted from the testing process of flight control software, and then the state transition model of flight control software was constructed. Base on the status transition model, a testing case chain based method for the reproduction of flight control software was proposed. Furthermore, with the restriction of the minimum cost, a method for finding the shortest path in the testing process of the failure reproduction was analyzed and conducted. Finally, an experiment was implemented to an open-source project (Ardupilot) using the proposed method. The result shows that, the proposed method can not only yield the shortest path from the process of the reproduction of failure, but also narrow the scope of the defect localization to some extent. Our work presents a way to improve the debugging efficiency and may shed light on the failure reproduction of the flight control software.**

***Keywords*—flight control software, testing case chain, state transition model, failure reproduction**

基于测试用例链的飞控软件失效复现方法

杜晓婷* 王楠

北京航空航天大学自动化科学与电气工程学院，北京 100191，中国

摘 要 软件失效复现在软件开发和维护过程中有非常重要的地位，软件的调试验证依赖于失效的复现。由于飞控软件对实时性要求高，控制算法复杂，导致飞控软件在开发和维护过程中都需要进行反复调试。 因此对于飞控软件来说，失效复现问题显得尤为重要。目前关于失效复现的研究，大都针对通用软件，且这些方法并不适用于飞控软件的失效复现，有必要对飞控软件的失效复现问题进行深入的研究。针对飞控软件测试过程的特点，将飞控软件的测试过程抽象为一条测试用例链。建立飞控软件状态转换模型，在此模型基础上提出了基于测试用例链的飞控软件失效复现方法，分析并给出了在最小代价下寻找失效复现的最短测试路径的方法。最后使用开源飞控项目 Ardupilot 进行失效复现实验，实验证明该方法不仅能给出失效复现的最短测试路径，而且在一定程度上缩小了缺陷定位的范围，对于提高调试效率有重要意义，为飞控软件的失效复现提供了一种可行的参考方法。

关键词 飞控软件，测试用例链，状态转换模型，失效复现

1. 引言

飞行控制软件是一类比较特殊的软件，与通用软件不同，飞控软件对安全性和实时性有极高的要求。飞行控制

*通讯作者, E-mail: xiaoting_2015@buaa.edu.cn

978-1-4673-8318-9/16/$31.00©2016 IEEE

系统可能会因为飞控软件故障向飞机舵机发出错误指令，也可能在机载设备发生故障时由于飞控软件未能及时进行故障处理，导致机毁人亡的事故发生。软件测试是保证和提高软件可靠性必不可少的主要措施之一。当一次测试失败后，调试人员需要重新运行该测试来复现失效，从而开展调试工作。而一次调试一般至少涉及到几次甚至几十次的失效复现，如果一次失效复现需要一天的时间，那么修复这个缺陷就需要几天甚至几十天的时间。可见，软件调试和修复的效率也依赖于失效复现的速率。目前很多研究者对自动化的失效复现技术进行了研究，这些方法大多是基于搜索的思想，通过收集发生失效时对象交互信息、执行数据、调用序列和堆栈等信息，并生成相应输入或测试用例，或者采用综合执行的方法来进行失效复现。这些方法通常需要消耗大量的资源，且复现率较低，不适用于飞控软件的失效复现。寻找一种针对飞控软件的自动失效复现方法，是本文所要解决的问题。

对于某一些缺陷而言，其失效复现不仅依赖程序当前的输入，还依赖程序当前的上下文。然而，创建这些上下文可能需要非常长的时间，这就使得一次失效复现的时间非常长。这种缺陷在飞控软件并不少见。比如，如果我们发现某飞行器在高空高速飞行过程中的日志总是损坏，但是在地面的时候却不会，那么这个高空高速的状态就是该缺陷触发的上下文。显然，为了复现该缺陷，我们首先要让飞行器进入高空高速的状态。于是每次复现该失效飞控软件都需要经历起飞，爬升，加速等等这些过程，而这些过程会耗费非常多的时间。如果我们能加速这些失效的复现过程，就能提高飞控软件的开发效率，降低维护成本，这显然是非常有意义的。为了解决这个问题，我们建立了飞控软件的状态转换模型，把失效复现问题转化为如何在状态转换图中寻找一条最短的测试路径的问题。本文还给出了在较短时间内找到失效复现的最短路径的系统策略，大大缩短了飞控软件失效复现的时间。为了验证方法的有效性，我们在开源飞控项目 Ardupilot 进行了实验。实验结果表明，本文提出的方法能够在较短的时间内找到失效复现的最短路径。

本文的贡献主要有：

(1) 建立了飞控软件测试过程的状态转换模型。

(2) 提出了一种寻找飞控软件失效复现最短测试路径的系统策略。

(3) 给出了飞控软件失效复现最短测试路径，减少调试时间，缩小缺陷定位的范围，从而提高软件调试和修复的效率。

文章剩余部分结构如下：第 2 节介绍相关研究；第 3 节建立状态转换模型；第 4 节介绍基于测试用例链的最短路径生成方法；第 5 节通过实验，验证方法的正确性和有效性；第 6 节是对本文的总结。

2．相关研究

软件失效复现对开发人员来说是最具挑战性的工作之一，通常要耗费大量的时间和资源，手工失效复现更是一项繁重的工作，许多研究人员对失效复现问题展开研究，提出了许多自动化失效复现方法。

Wei Jin[1]等提出了 BugRedux 方法，这种方法是失效复现的一项通用技术，它采用问题报告中故障执行时的动态执行数据，并使用这些数据模拟观察到的现场失效。这种方法能收集到 13.46%的对失效有用的信息。对于通用软件来说，这种方法是可接受的，可是对于飞控软件来说却是远远不够的。Cristian Zamfir 和 George Candea[2]也是利用缺陷报告中的信息，如核备份，堆栈轨迹信息等，通过“顺序路径合成技术”以及“线程调度合成技术”来复现缺陷。对于通用软件，开发者可以由用户反馈得到大量问题报告，从而收集有效信息，而飞控软件我们是无法得到大量问题报告的，所以以上方法并不适用。

由于问题报告中很少能提供足够的信息来复现失效，Fitsum Meshesha Kifetew[3][4]等提出了基于搜索的软件恢复方法，这种方法把失效复现问题看作一个搜索问题，复现一个失效看作是搜索一组导致失效路径执行的输入。通过对程序进行插桩，搜集一系列关于失效的运行时间信息或执行数据信息，产生可以实现失效复现的输入，这种方法的平均失效复现率为 79%。Ning Chen 和 Sung Kim[5]提出了 STAR 方法，通过收集失效堆栈轨迹，以堆栈轨迹作为动态信息来度量复现信息与原始信息的符合程度。这种方法能够收集到 42.3%的对原始失效有贡献的信息。Tobias Roehm[6][7][8]等通过记录用户与软件的交互操作轨迹信息，并利用信息缩减技术提取交互信息中的有效成分。

Shay Artzi 和 Sunghun Kim[9]等人记录了软件中的调用堆栈信息，根据该信息创建单元测试用例复现缺陷。文献[10]通过分析程序崩溃后的 dump 来复现失效。这些方法主要的问题是记录过程需要可观的资源，整个测试过程的资源消耗很大。

为了缩小搜索空间，降低复现成本，Martin Burger 和 Andreas Zeller[11]提出了 JINSI 方法。通过使用 delta debugging 和 slicing 的方法在保证失效被复现的情况下，减少内存中对象的交互次数，将所要搜索的源代码降低到 0.22%。但是搜索空间通常包含数百万的状态，每个状态又包含数千的变量，即使降低搜索空间后，工作量仍然是巨大的。

上述这些方法都是针对通用软件，由于飞控软件的特殊性，对飞控软件失效复现适用性并不好。首先，我们无法

得到大量的问题报告来收集信息。另外，飞控软件具有很高的复杂性，飞控软件的代码常达数百万甚至上千万行，基于搜索的方法对飞控软件来说显然是不现实的，或者需要消耗大量的资源。最关键的是，飞控软件作为安全性关键软件，控制算法复杂，实时性要求高，开发和维护过程中都需要进行反复调试，针对飞控软件设计一种高效且快速的失效复现方法就显得尤为重要。我们可以根据飞控软件的特点，将引起失效的测试用例链相关的所有测试用例和它们之间的关系用一个状态转换模型来表示，这部分内容在第三节阐述。

3．飞控软件状态转换模型

飞控软件的主要测试过程是定时执行一个主循环，每次循环中飞控系统都要接收传感器命令和驾驶指令，输出控制指令。每一次主循环都可以看作一个测试用例，输入是传感器信号和控制指令，输出是舵机角度，油门，写入的日志等。飞控软件的整个测试过程，就是一连串测试用例输入并执行的过程，我们把这个过程叫做一个测试用例链。在测试过程中，不同阶段输入并执行不同的测试用例，随着测试用例的运行，驱动飞控软件进入不同的状态。

我们用一个有向图来表示整个测试用例池和飞控软件可能到达的状态，以及它们之间的约束关系，如图 1 所示。这个有向图中，节点代表飞控软件的状态，边代表一个测试用例，边的权重就是这个测试用例消耗的时间，我们把该图叫作飞控软件测试过程的状态转换图。图中有两个特殊节点，开始节点和结束节点，分别用 S 和 E 表示。于是，一次测试完整的过程可以表示成状态转换图中从开始节点，经过若干边和若干节点，最终到达结束节点的一个路径。如果在一次测试中，飞控软件出现了失效，在失效处我们会终止测试。这种情况下，本次测试在状态转换图中所对应的路径终止于出现失效的状态，而不是结束节点，我们把这个状态叫做目标状态。现在，我们希望能够复现这个失效。

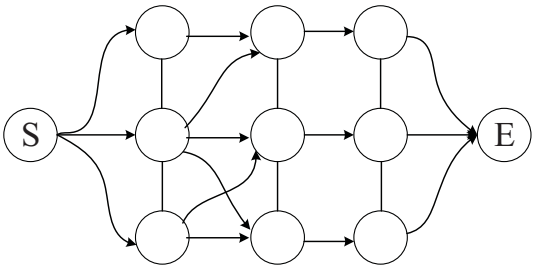

图1　表示测试用例池和飞控软件状态的状态转换图

如果失效总是能够在目标状态下触发，那么失效复现的目的很明确：尽快到达目标状态。对应在状态图下，就是找到一条从开始节点到目标状态的最短路径。进而，复现过程就是依次执行该最短路径上的每一条边所对应的测试用例。然而，在实际的测试过程中，失效不仅依赖于最终的失效状态，还可能依赖于某些中间状态，甚至是某些状态共同作用的结果。那么失效复现问题就转换成了寻找一条经过这些中间依赖状态，到达失效状态的路径的问题。最简单的复现策略就是完全按照原来的测试过程重新测试，以此来复现失效。在这种策略下，每次失效复现都要消耗原始测试所使用的时间。我们希望找到一种能够快速复现失效的策略，这也是第四节我们所要解决的问题。

4．基于测试用例链的飞控软件最短测试路径生成方法

4.1　基本假设

假设失效的出现不仅仅依赖于目标状态，还依赖于在此之前所经历过的某些状态。如果失效依赖N个先继状态，那么我们把这种依赖叫作N阶依赖。在N阶依赖的假设下，最优测试路径是从开始节点依次经过这N个依赖状态到目标状态的最短路径。

假设当前出现失效的测试用例总共有M个中间状态，即失效有可能依赖$1,2,\dots,M$个状态。不妨假设失效实际依赖n个状态，那么我们可以从这M个中间状态中选择n个状态，然后找到经过这n个状态的最短路径进行测试，如果最后出现了失效，则说明这n个状态就是该失效所依赖的状态，并且该最短路径就是最优的复现失效的路径，这样的选择有C_M^n个。如果考虑依赖的状态数可能是$1,2,\dots,M$，那么总共的候选路径有$C_M^1 + C_M^2 + \cdots + C_M^M$个。

从前面的分析可以看到，如果对所有的候选路径进行测试，我们就能找到耗时最短的测试且又能触发失效的测试路径，我们把这条路径叫作目标路径。但是由于候选路径太多，这样的策略是不现实的。因此，本部分提出的方法所解决的问题就是：如何以最小的测试代价在这些候选路径中找到目标路径。

4.2　基本定义

定义 1（最短测试路径）从开始节点依次经过所有依赖状态，最终达到目标状态的最短路径。

定义 2（失效路径）对于一个出现失效的测试用例链，可以表示为状态转换模型中的一条路径，起始节点是测试开始节点，终止节点是出现失效的节点，我们将这条路径叫做失效路径。

定义 3（候选路径）根据测试用例链所经过的中间状态，选出的所有可能的失效路径，即为候选路径。假设当前出现失效的测试用例总共有M个中间状态，那么失效有可

能依赖$1,2,\ldots,M$个状态。不妨假设失效实际依赖n个状态，$n=1$时，我们从M这个状态中选出 1 个状态作为必须经过的中间状态，这样我们得到了C_M^1个候选路径：$\{path(s,i_1,f),path(s,i_2,f),\ldots,path(s,i_M,f)\}$；$n=2$时，我们可以得到$C_M^2$个候选路径。当$n$取完$M$个值后我们得到了$C_M^1+C_M^2+\cdots+C_M^M$个候选路径。其中$path(i_1,i_2,i_3,\ldots,i_n)$表示一条从$i_1$开始，沿着最短路径依次经过$i_2,i_3,\ldots$,最后到达$i_n$的路径。

定义 4 （重复路径）由于图对路径的约束，得到的候选路径中有许多是重复的，称为重复路径。

定义 5 （路径包含）测试路径是否会引起失效这件事情并不是独立的，而是有关联的。比如，有一条测试路径是 A→B→C→D，而另一条测试路径是 A→B→D，可以看出它们是有关联的。在满足前文对失效的假设下，如果前者没有出现失效，后者一定也不会出现失效。相似的，如果后者出现了失效，那么前者一定也会出现失效。我们把这种关系叫路径的包含关系，在这里，路径 A→B→C→D 包含了路径 A→B→D。

4.3 基本思想及方法流程

基于以上定义，本部分提出基于测试用例链的最短路径生成方法。基本思想可以概括为两点：删除重复路径和识别路径包含关系。由于图的路径之间存在约束，所有候选路径中很多是重复的，因此需要去除重复路径。识别路径之间的包含关系，并根据这种关系安排测试顺序，就可以更快速的找到目标路径。当引起失效的测试用例链出现后，我们就可以使用该方法快速找到能引起失效的最短测试用例链，从而解决第三节中提出的问题。具体分为六个步骤，如图 2 所示，本小节我们将具体介绍前 5 个步骤。

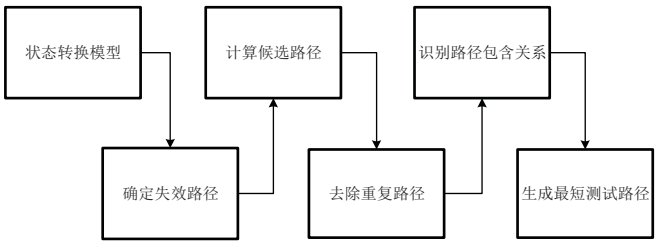

图2 基于测试用例链的最短路径生成步骤

(1) 建立状态转换模型及确定失效路径

我们要建立状态转换模型，并将失效的测试用例链表示为状态转换模型中的一条失效路径。如图 3 所示，当给出一个失效的测试用例链后，将它表示为状态转换模型中的一条失效路径。其中实心节点表示出现失效的节点，斜条纹节点表示测试经历的中间节点，加粗的边表示该测试用例链包含的测试用例。S 是测试开始节点，E 是测试结束节点。我们可以假设该图中直线边的权重都 1，曲线边的权重都为 1.5。当从 A 节点到 B 节点有边时，表示当飞控软件处于 A 状态时，可以运行 A→B 边所对应的测试用例。

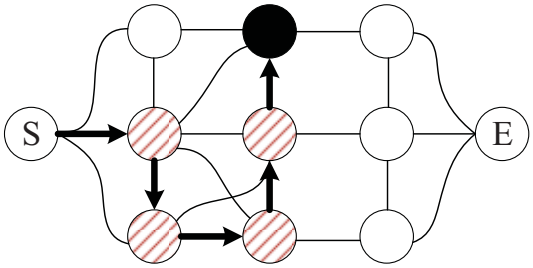

图3 原始出现失效的测试路径

(2) 计算候选路径

为了方便计算，我们假设失效最多依赖 2 个状态，可以得到在这种假设下所有的候选路径。图 4 是当失效依赖 1 个状态时所有可能的最短路径，图 5 是当失效依赖 2 个状态时所有可能的最短路径。其中竖线填充的节点是生成路径时我们假设失效所依赖的状态。

(3) 去除重复路径

在得到的所有候选路径中，有许多是重复的，这时需要先删除所有的重复路径。在图 4 和图 5 得到的候选路径中，c，j，f 这三条路径就是完全相同的，同样路径 i 和路径 b 也是相同的。去掉所有重复路径后，总的候选集如图 6 所示。

(4) 识别包含关系

在去掉所有重复路径的基础上，识别出所有的包含关系，并将这种关系表达成有向图的形式。其中节点表示候选路径，弧表示包含关系，当有从节点 A 到 B 的弧时，表示 A 包含了 B。图 7 用这种有向图表示了图 6 中的候选路径的包含关系。

本小节建立了状态转换模型，并将原始的失效测试用例链在状态转换模型中表示出来，通过去除重复路径减小总的路径候选集，识别候选路径的包含关系，是下一小节中生成最短测试路径的基础。

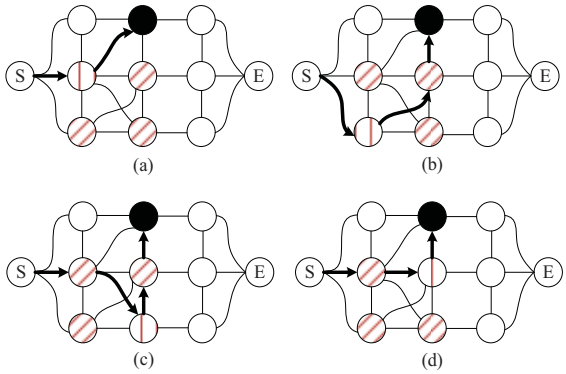

图4 当失效依赖 1 个状态时的候选集

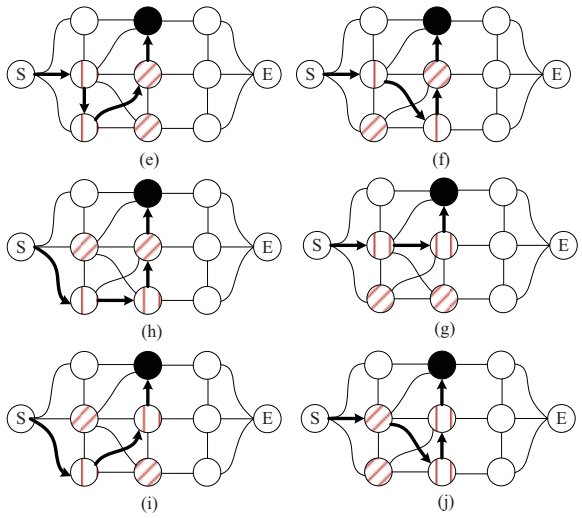

图5　当失效依赖 2 个状态时的候选集

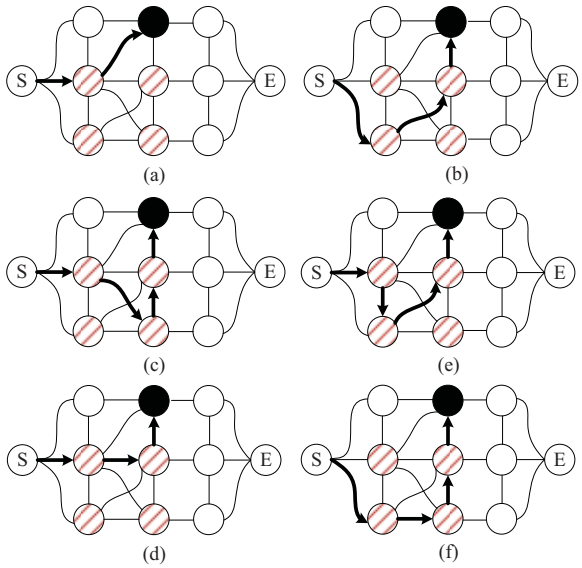

图6　去掉重复元素后的候选集

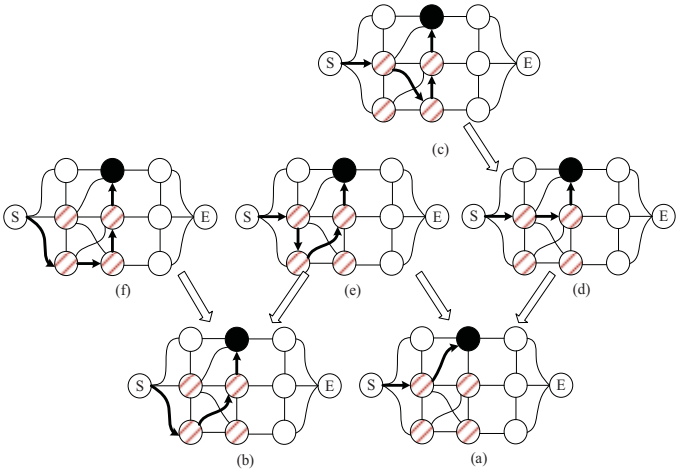

图7　表示候选路径包含关系的有向图

4.4　目标路径优化搜索技术

得到候选路径包含关系图后，下面我们将通过目标路径优化搜索技术以最小代价生成最短测试路径，该方法可以分为计算测试代价和搜索目标路径两部分，下面将做详细描述。

(1)　识别包含关系

用G表示上一步中的候选路径包含关系图，用$\mathbb{G}$表示G的所有子图集合。函数$F(X):\mathbb{G}\to R$表示在最坏的情况下，找到图X中目标路径所消耗代价的最小值。用$X(e)$表示节点 e 以及 e 的所有后继节点组成的子图。用$X/X(e)$表示从图X中去掉$X(e)$后的图。由包含关系的性质有：

$$F(X)=\min_{\forall e\in x}\Big(\max\big(F\big(X(e)\big),F\big(X/X(e)\big)\big)+F(e)\Big)$$

根据上述递归公式，我们就能解出测试代价上界的最小值。在每一步的递归中，我们需要记录 min 函数具体选择的节点，该节点就是在这一步中应该测试的节点。在图 7 中，假设现在我们选择对路径 e 进行测试，测试路径 e 所消耗的代价是 F(e)=1+1+1.5+1=4.5，接下来，我们计算剩下的测试需要消耗的代价。分两种情况讨论：

a)　路径 e 出现了失效

路径 e 已经经过了失效所依赖的所有状态，因此，目标路径一定被路径 e 所包含，也就是说，在图 7 中，从路径 e 出发，沿着表示包含关系的边，一定能够到达目标路径。因此，图中无法被路径 e 到达的节点就可以删除了。删除这些节点后得到$X(e)$，如图 8 所示。这种情况下，接下来的测试代价就是$F(X(e))$。

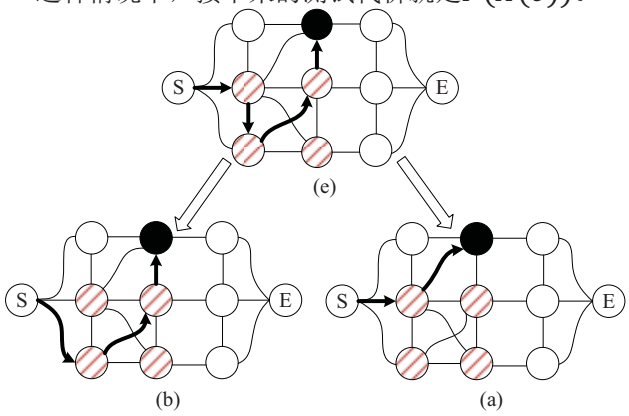

图8　假如路径 e 出现失效之后，候选集被缩小的情况

b)　路径 e 没有出现失效

这种情况与上面相反。路径 e 没有出现失效说明路径 e 并没有完全包含失效所依赖的状态，因此，目标路径一定不被 e 所包含。所以，我们可以将路径 e 和其所能够到达的节点从候选集中删除。删除这些节点后得

到$X/X(e)$，如图 9 所示。在这种情况下，接下来的测试代价就是$F(X/X(e))$。

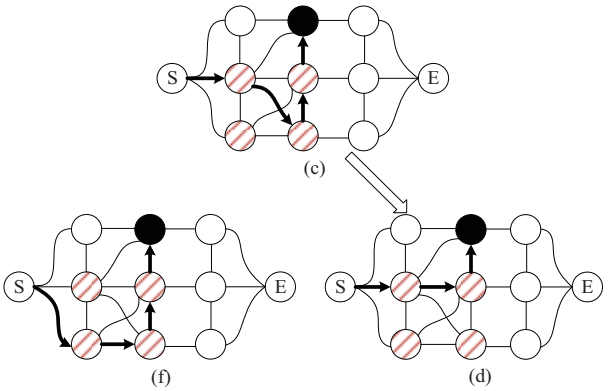

图9　假如路径 e 没有出现失效，候选集被缩小的情况

总的来说，如果我们选择首先测试路径 e 的总代价就是$\max(F(X(e),F(X/X(e)))+4.5$。通过递归的求解$F$，我们就可以算出首先测试路径 e 的最坏代价。通过遍历所有的路径，我们就能计算出$F(X)$的值。

(2)　搜索目标路径

通过记录对$F(X)$递归求解过程中每一步$\min$函数所选择的节点（用$\min(X)$表示），我们能够生成一棵由测试路径组成的决策树。其中$\min(X)$是父节点；$\min(X(e))$是左孩子，表示测试出现失效时的路径；$\min(X/X(e))$是右孩子，表示测试没有出现失效的路径。

在求解过程中，$F(X)$表示从X中找到目标路径上界的最小值。显然，如果根据该决策树来寻找目标路径，就能在最坏的情况下，以最小的代价找到目标路径。接下来，我们将通过实验验证该方法的有效性。

5．实验

在这一节中，我们应用本文提出的方法在实际飞控软件上进行实验，验证方法的正确性和有效性。该实验中的各个测试项目来源于开源飞控项目 Ardupilot 的各项功能，各个项目测试时间是本文给出的一个估计。首先我们对实验对象以及实验设计与实现过程进行描述，然后对实验结果进行分析。

5.1　实验对象

Ardupilot 是一套无人机自动飞行控制系统，支持多旋翼飞行器，固定翼飞机和传统直升机。系统由固定翼/多旋翼载机，APM 飞行控制板，数传模块，电子罗盘，各类传感器和地面站组成。飞机固件使用 mavlink 协议，支持多种地面站。虽然不如载人飞行器的飞控软件复杂，但是它也有近 20 万行代码（不包括实时操作系统 Nuttx 的代码和底层驱动的代码），足够满足本次实验的要求。本实验中的方法使用 Python 语言编程实现。

5.2　实验设计与实现

假设在一次测试过程中，我们依次进行了以下测试：起飞，爬升 100m，向北 100m，GPS 定点测试，PID 参数自动调整测试，PID 参数回传测试，mavlink 通信测试。在最后一次测试中，即 mavlink 通信测试中，发生了失效。这个测试过程可以用图 10 表达。

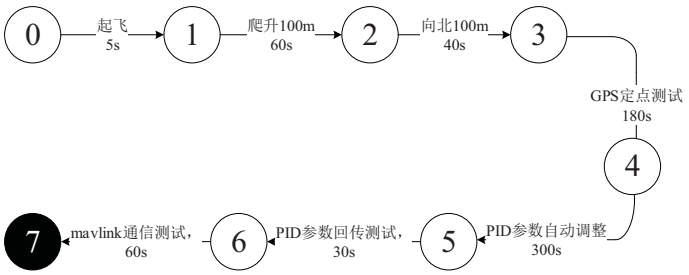

图10 发现失效的测试过程

现在，我们已知图 10 中测试过程，并已知在 mavlink 通信测试时，发生了失效，希望复现这个失效。我们可以选择直接进行 mavlink 通信测试，但是这样测试并不一定会发生失效，因为失效很可能依赖之前所运行的某些测试，比如有可能是 PID 参数回传测试后，由于软件的错误，使通信模块处于了一种不可工作的状态，同理，也有可能是 GPS 定点测试对 mavlink 通信测试发生了影响。更复杂的情况是，失效有可能是这两者共同作用的结果。

我们也可以选择重复整个发生失效的测试过程，但是这样也许会运行许多不必要的测试用例，浪费了许多时间。更重要的是，在调试过程中，我们往往需要多次复现失效，这样会浪费更多的时间，过长的测试过程也不便于我们对失效的原因进行分析。下面我们使用本文提出的方法来寻找能最快触发失效的测试用例链。

首先，添加测试用例池中与这些测试相关的其他测试，比如我们可以添加爬升 10m 和向北 10m 这两个测试用例，如图 11 所示。

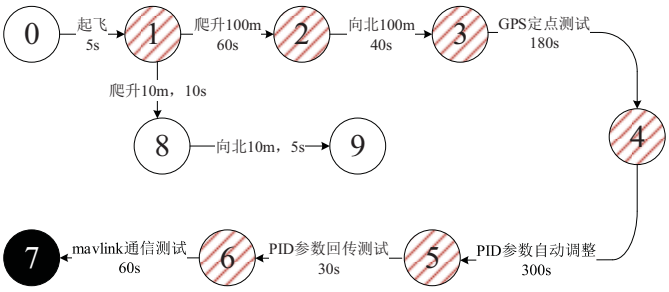

图11 补充相关测试状态后的测试过程

接下来，我们需要补全测试过程之间的约束。对于某个状态，有一些测试用例是可以输入的，另一些在该状态下则是不可输入的。比如在空中的时候我们可以收起起落架，而在地面就不可以，所以我们要先确定哪些测试用例是可以相邻执行的。比如，在执行完爬升 100m 后，我们可以直接执行 GPS 定点测试，因此，我们使用一条弧将图 11 的状态 2 和状态 4 连起来。相反，PID 参数回传测试必须在 PID 参数自动调整后执行，因此，其他状态不能直接连接到图 11 的状态 6。由于连接完成之后的弧非常多，在图中不易表示清楚，我们把连接完成的有向图用邻接矩阵表示，如表 1 所示。表中的 i，j 位置表示从状态 i 到 j 所用的测试用例的持续时间，即测试代价。表中的空白表示从 i，j 没有直接相连的弧。

表1　测试用例依赖关系图的邻接矩阵

	0	1	2	3	4	5	6	7	8	9
0	0	5								
1		0	60					60	10	
2			0	40	180	300		60		5
3				0	180	300		60		
4					0	300		60		
5						0	30	60		
6							0	60		
7								0		
8				40	180	300		60	0	5
9					180	300		60		0

求解的结果是一棵由测试路径组成的决策树，如图 12 所示。树中节点的编号是测试路径的编号（-1 代表没有不存在的路径），每个编号所对应的测试路径具体所包含的测试用例和测试代价如表 2 所示。该决策树可以指导我们如何最快的找到能够引起失效的测试用例。从决策树的根节点开始（也就是测试路径 6），选择将该节点所对应的测试路径进行测试，如果测试通过，继续向下，选择 P 分支的节点；如果测试失败，则选择 F 分支的节点。持续这样的过程，直到测到叶子节点，该叶子节点就是我们找到的最短测试路径，叶子节点对应的测试路径和测试代价在表 2 中用深色部分标出。

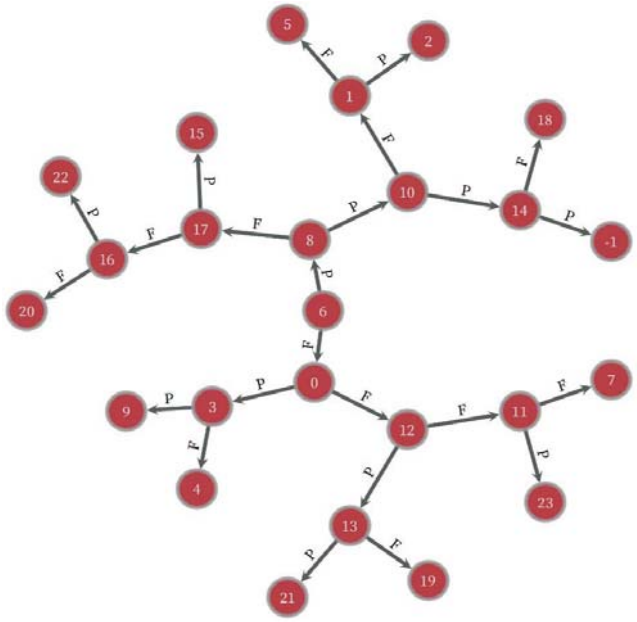

图12 解出的决策树

5.3　实验结果分析

表 2 中深色部分给出了失效复现的最短测试路径的编号、依次经过的状态和测试代价。通过这个表，我们可以确定失效复现的最短测试路径，从而避免了每次调试都要按原测试路径进行，在飞控软件的反复调试过程中，节省了大量的时间。我们还可以根据该最短测试路径，缩小缺陷定位的范围，有效提高调试效率。本文的失效复现方法同时给出了整个测试过程的上界：3075 秒。即，我们最多需要使用 3075 秒就可以找到最短测试路径。如果我们使用枚举的方式进行测试，需要的时间上界是：28715 秒。相比枚举方法，我们的方法只使用了大约 1/10 的时间。

6．结论

本文基于飞控软件的测试特点，将飞控软件的测试过程抽象成一条测试用例链。建立了飞控软件的状态转换模型，将失效的测试用例链表示为状态转换模型中的一条失效路径，从而将失效复现问题转换成寻找状态转换模型中的一条最短测试路径。通过去除重复路径和识别路径包含关系，并使用本文提出的目标路径优化搜索技术找到了最短的失效复现路径。此外，以 Ardupilot 为实验对象，本文设计并进行了实验。结果表明，使用目标路径优化搜索技术能够有效的提高寻找最短测试路径的速度，从而提高飞控软件失效复现的速度。寻找到的最短测试路径不仅可以避免每次调试都按原测试路径进行，而且可以缩小缺陷定位的范围，对于提高飞控软件调试和修复效率有很大意义。对于确定能够被复现的失效，通过本文的方法能够有效加

快失效复现的速度。由于飞控软件及其运行环境的复杂性，有许多失效仍然是无法被确定复现的，未来我们将在本文基础上对无法被确定复现的失效进行更多研究。

表2　候选测试路径

测试路径编号	依次经过的状态	测试代价
0	0 1 2 4 5 7	605.0
1	0 1 2 3 4 7	345.0
2	0 1 8 3 4 5 7	595.0
3	0 1 8 4 5 6 7	585.0
4	0 1 8 5 6 7	405.0
5	0 1 8 3 4 7	295.0
6	0 1 2 4 5 6 7	635.0
7	0 1 7	65.0
8	0 1 2 3 5 6 7	495.0
9	0 1 2 5 6 7	455.0
10	0 1 2 3 4 5 7	645.0
11	0 1 8 5 7	375.0
12	0 1 8 4 5 7	555.0
13	0 1 2 4 7	305.0
14	0 1 2 3 4 5 6 7	675.0
15	0 1 8 3 5 6 7	445.0
16	0 1 2 3 7	165.0
17	0 1 2 3 5 7	465.0
18	0 1 8 3 4 5 6 7	625.0
19	0 1 2 7	125.0
20	0 1 8 3 7	115.0
21	0 1 2 5 7	425.0
22	0 1 8 3 5 7	415.0
23	0 1 8 4 7	255.0

参考文献

[1] Jin, W., & Orso, A. BugRedux: reproducing field failures for in-house debugging. In Proceedings of the 34th International Conference on Software Engineering (pp. 474-484). IEEE Press, 2012.

[2] Zamfir, C., & Candea, G. Execution synthesis: a technique for automated software debugging. In Proceedings of the 5th European conference on Computer systems (pp. 321-334). ACM, 2010.

[3] Kifetew, F. M., Jin, W., Tiella, R., Orso, A., & Tonella, P. Reproducing field failures for programs with complex grammar-based input. In *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on* (pp. 163-172). IEEE, 2014.

[4] Kifetew, F. M. A search-based framework for failure reproduction. In *Search Based Software Engineering* (pp. 279-284). Springer Berlin Heidelberg, 2012.

[5] Chen, N., & Kim, S. STAR: stack trace based automatic crash reproduction via symbolic execution. *Software Engineering, IEEE Transactions on*, *41*(2), 198-220, 2015.

[6] Roehm, T., & Bruegge, B. Reproducing software failures by exploiting the action history of undo features. In *Companion Proceedings of the 36th International Conference on Software Engineering* (pp. 496-499). ACM, 2014.

[7] Roehm, T., Gurbanova, N., Bruegge, B., Joubert, C., & Maalej, W. Monitoring user interactions for supporting failure reproduction. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on* (pp. 73-82). IEEE, 2013.

[8] Roehm, T., Nosovic, S., & Bruegge, B. Automated extraction of failure reproduction steps from user interaction traces. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on* (pp. 121-130). IEEE, 2015.

[9] Artzi, S., Kim, S., & Ernst, M. D. Recrash: Making software failures reproducible by preserving object states. In *ECOOP 2008–Object-Oriented Programming* (pp. 542-565). Springer Berlin Heidelberg, 2008.

[10] Weeratunge, D., Zhang, X., & Jagannathan, S. Analyzing multicore dumps to facilitate concurrency bug reproduction. *ACM Sigplan Notices*, *45*(3), 155-166, 2010.

[11] Burger, M., & Zeller, A. Minimizing reproduction of software failures. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (pp. 221-231). ACM, 2011.