程序错误的形式化定义和程序容错能力的分级

刘晓建[1], 朱智林[2], 杜慧秋[2]

1. 西安科技大学 计算机学院, 西安 710054
2. 山东工商学院 信息与电子工程学院, 山东 烟台 264005

摘 要：针对程序失效相关概念的形式化定义和程序容错能力的分级, 分析了程序缺陷、状态偏差以及程序失效等基本概念之间的差异, 并在基于状态的程序行为理论的框架下, 形式化定义了这些概念. 从程序安全和活性性质的可满足性方面, 给出了一个程序容错能力的分级方案, 有助于相关概念的准确理解以及系统对现有方法容错能力的区分.

关键词：程序失效; 程序语义; 容错; 形式化方法; 软件安全性
中图分类号：TP311 **文献标识码：**A **文章编号：**0455-2059(2016)04-0001-08
DOI: 10.13885/j.issn.0455-2059.2016.04.001

Formal definition of program faults and hierarchy of program fault-tolerant abilities

Liu Xiao-jian[1], Zhu Zhi-lin[2], Du Hui-qiu[2]

1. College of Computer Science and Technology, Xi'an University of Science and Technology, Xi'an 710054, China
2. School of Information and Electronic Engineering, Shandong Institute of Business and Technology, Yantai 264005, Shandong, China

Abstract: Two issues were addressed: the formal definitions of the concepts relevant to program faults, and the comparison and classification of program fault-tolerant abilities. First, the subtle differences were analyzed between these basic concepts: faults, errors and failures, and represented their formal definitions by using the state-based theory of program behavior; then a hierarchy for software fault-tolerant abilities was proposed from the aspect of satisfying the safety and liveness properties. The main purpose of this work was to facilitate an accurate understanding of basic concepts and classifying fault-tolerant abilities of different approaches in a systematic way.

Key words: program fault; program semantic; fault-tolerance; formal method; software safety

有关程序出错行为和错误处理方法的研究一直是软件工程研究和实践的重要内容之一. 通常倾向于将程序错误看作是一种可以被完全隔离和消除的负面因素, 然而由软件工程的理论和实践可知, 程序错误不可能被完全消除, 程序的正确性都是相对的, 程序的可接受性总是限定在一定条件下的. 因此需要从新的角度来看待程序与程序错误之间的关系, 即并不将程序错误看作独立于程序之外的因素, 而是将其看作程序自身固有的成分. 程序的生存演化过程就是不断克服错误造

收稿日期：2015-03-21 **修回日期：**2015-05-11
基金项目：陕西省教育厅科研计划项目(2013JK1188); 山东省自然科学基金项目(ZR2012FL11); 西安科技大学博士后启动基金项目(2013QDJ023); 国家自然科学基金煤炭联合基金项目(U1261114)
作者简介：刘晓建(1971−), 男, 陕西西安人, 副教授，博士, e-mail: 780209965@qq.com, 研究方向为软件形式化方法.

成的负面影响、实现和逼近设计目标的动态发展过程.

Avizienis 等[1]已明确定义了程序错误相关的几个概念, 分析了与程序的出错行为相关的几个重要性质(如可靠性、安全性、可依赖性、可接受性等). 在错误处理方法上, 异常处理和断言法已在程序设计语言层面被普遍采用. 恢复块、N版本程序和N版本自检程序等已发展为容错软件设计的典型方法[2], 在程序安全领域得到了广泛应用. Cristian[3]给出了异常处理机制的形式语义, 并采用Hoare 逻辑对软件的出错行为进行演绎. Liu 等[4]基于 Action system, 使用程序变换理论研究了缺陷对程序行为的影响. Bernardeschi 等[5]采用进程代数, 研究了环境缺陷与程序行为的关系. Reese[6]提出了软件偏差分析方法, 分析软件偏差的传播方式及其影响.

随着软件应用领域的不断深入, 对软件行为和错误处理能力提出了一系列新的需求, 如抗衰以及再生性[7-8]、降级模式[9]、程序健康性[10]、自愈性[11-12]、老化等. 这些需求都与程序自身以及外部环境的各种缺陷密切相关, 如何以新方法处理这些缺陷目前是一项重要挑战. 由于开源软件以及软件库的出现, 使得程序的缺陷以及发现缺陷的过程得到了跟踪和记录, 这就为使用数据挖掘和统计方法研究程序的失效模式和缺陷模型, 揭示程序失效机制的规律提供了可能[13-14].

尽管相关研究已较为成熟, 但有关程序错误相关概念的形式化定义却鲜有提及, 而关于程序容错能力的分级通常也零散的出现在不同文献中, 难以形成理论化、系统化的体系. 本研究采用基于状态的程序行为理论, 形式定义和解释了程序错误的基本概念, 给出了程序容错能力的层级分类, 为准确理解相关概念, 系统认识程序出错行为与环境的关系做出了必要的理论铺垫.

1 相关概念

首先辨析与程序错误相关的几个主要概念, 然后采用程序的行为模型对这些概念进行定义和解释, 试图在统一的模型下描述程序的出错行为.

与程序错误相关的几个容易混淆的概念是: fault/defect (“缺陷”)、error (“偏差”)和 failure(“失效”). 有关它们的中文翻译在不同的文献中不尽相同, 此处忽略具体翻译上的不同, 更注重这些概念的内涵, 以下定义来自文献[1].

1) 失效: 程序失效是针对程序所提供的服务而言的, 指“程序所提供的服务与需求规格说明所规定的正确服务之间出现了偏差的现象”.

程序失效具有可观察性和危害性. 由于服务是用户可观察的状态序列, 因此服务的偏差能够为用户所察觉和感知. 如果程序失效不能被及时发现并控制, 那么失效有可能通过软硬件接口使硬件发生误动或失效, 造成程度不同的危害, 通常用危害等级和危害发生的概率来衡量[15].

2) 偏差: 程序偏差是针对程序状态而言的: “程序运行过程中, 实际运行状态相对于正确状态发生偏离的现象”.

由于程序状态分为内部状态和外部状态, 如果发生偏差的状态是内部状态, 则用户无法观察到偏差的发生; 如果发生偏差的状态是外部状态, 则用户可以观察到, 这样的偏差实际上就是失效. 偏差具有传递性, 内部状态的偏差会随程序的运行不断传播和扩大, 直至传播到外部状态, 发生失效. 从这个意义上讲, 程序偏差是引发程序失效的原因.

另外, 状态偏差是相比“正确状态”而言的, 然而如何获得正确状态却并不直观. 程序的规格说明通常不可执行, 而且具有一定的抽象性, 难以规定什么是正确的程序内部状态. 因此, 一般情况下总是假定存在一个先验的观察者能够知道什么是正确的状态, 并判断状态偏差.

3) 缺陷: “缺陷”在不同上下文中有不同的英文与之对应. 例如将存在于程序代码中的缺陷称为 bug, 将存在于需求规约中的缺陷称为“漏洞”(defect 或 flaw), 硬件中的缺陷通常称为 deficiency, 而由人所引起的缺陷常称为“错误”(errata). 程序缺陷是指存在于程序本身或程序所在环境中的、可能导致程序状态发生偏差的因素.

根据定义, 缺陷分为内部缺陷(存在于程序自身中的缺陷)和外部缺陷(通过输入接口传入或存在于程序环境中的缺陷). 程序中存在的 bug 是内部缺陷, 有可能导致程序的状态发生偏差; 计算机内存受到电磁干扰或辐射的影响, 使得存储的数据位发生极性反转, 导致程序状态发生了偏差, 这种极性反转对于程序而言就是一个外部缺陷. 外部缺陷通过内部缺陷起作用, 例如木马程序利用程序中的缓冲区溢出漏洞, 越权访问缓冲区中的数据, 造成数据泄露, 木马程序就是外部缺陷, 缓冲区溢出是内部缺陷, 木马程序利用了程序自身

存在的缺陷发生作用.

缺陷是引起状态偏差的原因, 但只有在特定条件下, 当缺陷被激活时才会引发偏差, 因此缺陷具有两种状态: 休眠状态和活跃状态, 即使处于休眠状态也不能否定缺陷的存在. 因此缺陷的存在是静态的, 不依赖于程序的使用方式.

缺陷、偏差和失效是相互关联的, 形成了一个因果关系链——威胁链(图 1).

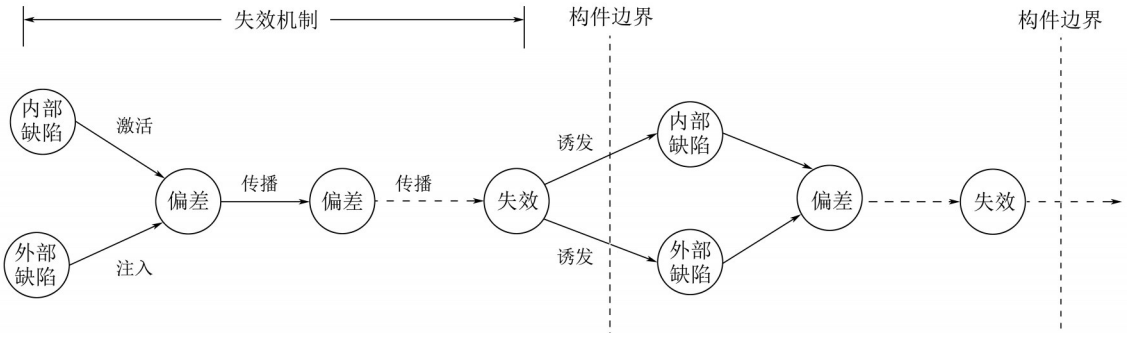

图 1　威胁链

Fig. 1　Threat chain

程序的内部缺陷被激活或外部缺陷注入后, 会产生状态偏差, 偏差的传播导致该构件失效. 值得注意的是, “缺陷→偏差→失效”这个因果链局限在同一个构件边界内, 而“失效→缺陷”因果链却跨越了不同构件的边界, 这就意味着, 一个构件的失效成为另一个构件缺陷的原因. 所谓“失效机制”就是分析和探究失效发生的过程和机理, 即探究从缺陷、偏差直到失效发生的各个环节及其因果关系.

2　程序行为模型

上一节给出了程序缺陷、偏差和失效的定义, 但是如何准确的理解这些概念, 甚至进一步理论分析和推理程序的出错行为, 仅靠这些文字定义是无法解决的. 因此, 有必要在程序语义的基础上讨论这些概念. 本节首先给出程序的行为模型, 然后利用此模型讨论这些概念的形式化定义.

通常一个程序是由若干构件通过特定的方式组织在一起而形成的. 程序所呈现的错误行为通常都是由它所包含的构件的错误行为通过特定的传播方式而引起的, 因此需要在构件层次上定义程序的行为.

一个构件是一个程序(顺序或并发)program, 而程序由程序变量的有限集合 PROGVAR 和操作的有限集合 ACTIONS 构成, 即

$$\text{Program} = \langle \text{PROGVAR}, \text{ACTIONS} \rangle$$

操作 action ∈ ACTIONS 是一个形如“guard→command”的卫士命令[16], 其中 guard 是程序变量的一个布尔表达式, 称为操作的“卫士条件”, command 是一个语句, 称为“命令”. 如果一个操作的卫士条件在某个状态上为真, 则称该操作是“可执行的”. 基于上述构件程序 Program, 给出以下基本概念.

1) 状态变量

构件状态变量集合 VAR 是名字的有限集合. 一个状态变量具有一个特定的类型和取值范围. 集合 VAR 中所有变量的值集记为 VAL. 构件状态变量既包括程序变量 PROGVAR 也包括其他拟观察的变量 OBS, 即

$$\text{VAR} = \text{PROGVAR} \cup \text{OBS}.$$

拟观察变量的选取依赖于具体问题: 如果拟观察构件行为是否终止, 则需要增加变量 o_k 和 o_k'; 如果拟观察构件运行的时间特性, 则需增加时钟变量 T_{tick}; 如果拟观察运行过程中构件结构的变化, 则需增加描述构件连接状态的变量; 而如果拟观察运行中功耗和存储空间的变化, 则需增加相应的变量.

2) 内部变量和外部变量

状态变量分为内部状态变量和外部状态变量, 即

$$\text{VAR} = \text{INTVAR} \cup \text{EXTVAR},$$

$$\text{INTVAR} \cap \text{EXTVAR} = \varphi.$$

内部状态变量 INTVAR 通常是程序变量中那些在构件之外不可观察的变量, 而外部状态变量 EXTVAR 通常包括程序变量中那些在构件外可观察的以及拟观察的变量 OBS, 即 OBS⊆EXTVAR. 外部变量是构件向外提供各种服务, 或其他构件观察该构件内部状态的唯一途径.

3) 程序状态

程序状态s是定义在构件状态变量集合VAR上的一个函数s:VAR→VAL, 可将每个状态变量映射为一个唯一值. 对于状态变量x∈VAR, 用$s[x]$表示状态s下x的值. 所有可能状态的集合记为$\sum$, 称为该构件的状态空间.

4) 状态谓词

简称谓词, 是在程序状态s上进行解释的谓词. 谓词P在状态s上的值用$s[P]$来表示. 如果$s[P]$为真, 则称s满足P, 或P在s上成立. P的特征集合, 记为S_P, 是满足P的所有状态的集合, 即$S_P=\{s|s[P]=\text{True}\}$.

定义1(程序行为) 程序Program的行为可以用程序的执行或程序状态的无限序列$\sigma=\langle s_0, s_1, s_2,\cdots\rangle$来刻画, 该序列满足以下条件

(i) 任给σ中的状态偶对$\langle s_i, s_{i+1}\rangle$, 存在一个操作guard→command∈ACTIONS在状态s_i上是可执行的, 且执行command之后将状态s_i迁移到状态s_{i+1}, 将这种迁移关系记为$s_i \xrightarrow{\text{command}} s_{i+1}$.

(ii) σ满足最大性, 即σ要么无限长, 要么最后一个状态是死锁状态, 即在最后一个状态上没有任何一个操作可以执行.

(iii) σ满足公平性, 即若某个操作沿着σ一直是可执行的, 则其最终将会被选择执行.

对于$\sigma=\langle s_0, s_1, s_2, \cdots\rangle$, s_0称为初始状态, $\langle s_i, s_{i+1}\rangle$称为$\sigma$中的一个“状态迁移步”(以下简称“迁移步”). 记状态的无限长序列的集合为$\sum^{\infty}$, 则σ∈$\sum^{\infty}$.

若程序是激发响应式的, 则需要不间断的对来自环境的激励做出响应, 其行为可以认为是无限长的; 如果程序是一个算法, 假如它的运行是发散的, 那么它的行为也是无限长的, 假如算法成功终止或进入死锁状态, 那么行为是有限长的, 这种情况可以认为末状态无限重复出现, 使得其行为也是无限长的.

定义2(X-Stutter迁移步[17]) 设$X\subseteq$VAR是变量集合, 如果迁移步$\langle s_i, s_{i+1}\rangle$不改变$X$中变量的值, 即$\forall x\in X\cdot s_i[x]=s_{i+1}[x]$, 则称$\langle s_i, s_{i+1}\rangle$为关于$X$的Stutter迁移步; 如果$\langle s_i, s_{i+1}\rangle$改变了$X$中任一变量的值, 即$\exists x\in X\cdot s_i[x]\neq s_{i+1}[x]$, 则称$\langle s_i, s_{i+1}\rangle$为关于$X$的非Stutter迁移步.

如果令X=EXTVAR, 则关于X的Stutter迁移步就是那些不改变外部程序变量和拟观察变量的迁移, 即该状态的迁移是不可观察的, 也就是内部状态迁移. 用户一般对内部状态迁移不感兴趣, 因此希望把这些Stutter迁移步抽象掉, 只保留用户可观察的迁移步. 于是引入下面的X-Stutter闭包这一概念.

定义3(X-Stutter闭包) 设$\sigma=\langle s_0, s_1, s_2,\cdots\rangle$是程序的一个状态序列, $\langle s_i, s_j\rangle$是一个关于X的Stutter闭包, 当且仅当

1) $\langle s_i, s_j\rangle$是σ中关于X的一个非Stutter迁移步;

2) 存在有限个状态$s_{i+1}, s_{i+2}, \cdots, s_{i+n}$, 使得$\langle s_{i-1}, s_i\rangle$是一个关于$X$的非Stutter步, $\langle s_i, s_{i+1}\rangle$, $\langle s_{i+1}, s_{i+2}\rangle, \cdots, \langle s_{i+n}, s_j\rangle$是$\sigma$中关于$X$的Stutter步, 而且$\langle s_{i+n}, s_j\rangle$是关于$X$的非Stutter步. 这种情况下, 称状态$s_{i+1}, s_{i+2}, \cdots, s_{i+n}$为“内部状态”(图2).

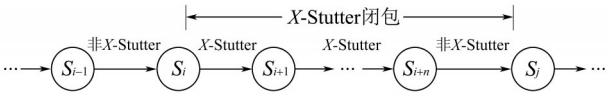

图2 X-Stutter闭包

Fig. 2 X-Stutter closure

显然, 如果令X=EXTVAR, 则X-Stutter闭包将s_i和s_j之间的内部状态和内部迁移抽象掉, 只保留了用户可观察的迁移步$\langle s_i, s_j\rangle$, 因此也将其称为“宏步”. 再从精化的角度看, 中间迁移$\langle s_i, s_{i+1}\rangle$, $\langle s_{i+1}, s_{i+2}\rangle, \cdots, \langle s_{i+n}, s_j\rangle$可以看作是对$\langle s_i, s_j\rangle$的细化和实现, 因此这些迁移也称为“微步”.

引入上面定义1–3的目的是为了定义“服务”的概念.

定义4(服务) 设$\sigma=\langle s_0, s_1, s_2,\cdots\rangle$为构件程序的行为, 则构件向用户提供的服务service是σ中用户可观察(或感兴趣)的状态序列, 即

$service=\langle s^0, s^1, s^2,...\rangle$, 满足$\forall i\geq 0\cdot\langle s^i, s^{i+1}\rangle$是$\sigma$中关于EXTVAR的Stutter闭包, 即服务就是由行为中那些用户可观察的状态迁移所构成的状态序列, 这样的状态s^i也称为“外部状态”.

3 形式定义

本节从程序的行为模型出发, 定义和解释软件偏差、失效和缺陷, 使软件的正常行为和出错行为能够被纳入统一的理论框架内进行讨论.

3.1 状态偏差

状态偏差是在软件运行过程中, 实际运行状

态相对于正确状态发生偏离的现象. 用距离作为状态之间偏离程度的度量, 当距离超过某个阈值时, 就认为状态发生了偏差.

定义 5(状态偏差) 设 s, s': VAR→VAL 是两个状态, 用 $d(s, s')$表示两个状态间的距离. 对于给定的偏差向量 $\vec{\delta}$, 如果 $d(s, s')\leqslant\vec{\delta}$, 则称 s 与 s'是相同的或一致的, 记为 $s=s'$; 反之, 如果 $d(s, s')>\vec{\delta}$, 称状态 s'偏离了状态 s, 记为 $s\neq s'$.

值得注意的是, 距离 $d(s, s')$的具体数学定义和 $\vec{\delta}$ 的选取依赖于变量集和具体问题.

例 1 设变量集为 VAR={x: BOOL, t: TIME, m: MEMORY}, 其中 x 为布尔类型的程序变量, t 和 m 分别表示程序执行时间和内存占用的变量, 则可用如下定义作为状态距离的度量.

内存单位是MB还是Mb?

$$d(s, s')=\begin{pmatrix}|s[x]-s'[x]\|\\|s[t]-s'[t]\|\\|s[m]-s'[m]\|\end{pmatrix}.$$

如果取 $\vec{\delta}=(0, 500\ \text{ms}, 1\ \text{Mb})^{\mathrm{T}}$, 则表示如果 s 和 s'的 x 值相同, 当系统处于这两个状态的时间和占用的内存相差分别不超过 500 ms 和 1 Mb 时, 则认为这两个状态相等或一致.

如果只想考查某些状态变量的偏差, 则需要对上述定义作一些限制.

定义 6(状态偏差的投影) 设 $X\subseteq$VAR 是感兴趣的状态变量集合, 用 $s|_X$表示状态在 X 上的投影(或限制). 如果 $d(s|_X, s'|_X)\leqslant\vec{\delta}|_X$, 则称 s 与 s'在变量集合 X 上是相同的(一致的), 记为 $s|_X=s'|_X$; 反之, 如果 $d(s|_X, s'|_X)>\vec{\delta}|_X$, 称状态 s'在变量集合 X 上偏离了状态 s, 记为 $s|_X\neq s'|_X$.

当某状态发生偏差之后, 该偏差通常会随着程序的运行不断传播, 导致后续的状态都发生偏差, 即软件的行为发生了偏差. 如果用状态序列中首次出现偏差的状态作为度量行为偏差程度的指标, 则可以得到关于行为偏差距离的定义.

定义 7(行为偏差及偏差传播路径) 设 $\sigma=\langle s_0, s_1, s_2, \cdots\rangle$ 和 $\sigma'=\langle s'_0, s'_1, s'_2, \cdots\rangle$ 是两个行为, 则 σ 和 σ'的偏差可以用距离 $d(\sigma, \sigma')=2^{-n}$ 来定义, 其中 n 是首次出现状态偏差的状态序号, 即对于任意 $i<n$, $s_i=s'_i$, 而且 $s_n\neq s'_n$. 若用 σ 表示正确的行为, 那么 σ'的后缀 $\langle s'_n, s'_{n+1}, \cdots\rangle$ 就表示状态偏差的传播路径.

显然, 两种极端情况是: 如果 σ 和 σ'完全相同, 则 n 趋近于∞, $d(\sigma, \sigma')$趋近于 0; 如果从起始状态开始就不相同, 即 $n=0$, 则 $d(\sigma, \sigma')=1$. 一般情况下 σ 和 σ'的距离落在[0, 1]之间.

3.2 失效

失效是可观察状态的偏离, 即当外部变量的值发生偏离时, 软件系统就发生了失效.

定义 8(失效) 设 $\sigma=\langle s_0, s_1, s_2, \cdots\rangle$ 和 $\sigma'=\langle s'_0 s'_2, \cdots\rangle$ 分别表示正确行为和对应的实际行为, service $=\langle s^0, s^1, s^2, \cdots\rangle$ 和 service$'=\langle s'^0, s'^1, s'^2, \cdots\rangle$ 分别是由 σ 和 σ'提供的服务. 如果存在 $i\geqslant 0$, 使得 s^i 和 s'^i在外部变量集 EXTVAR 上发生了偏差, 即 $s^i|_{\text{EXTVAR}}\neq s'^i|_{\text{EXTVAR}}$, 则称 service$'$发生了失效.

3.3 缺陷

软件缺陷是诱发状态偏差的原因, 无论是内部缺陷还是外部缺陷, 都可以将其抽象为某种操作, 并加入到正常程序中, 形成一个能够模拟真实缺陷环境的程序. 由于缺陷的发生是随机的, 因此还需要增加描述缺陷发生概率特征的函数.

作了这样扩充的程序变为:

$$\text{Program}_F=\langle\text{PROGVAR, ACTIONS, ACTIONS}_F, F(t, w)\rangle.$$

其中 ACTIONS 是正常操作的集合, ACTIONS$_F$是缺陷操作的集合, $F(t, w)$是缺陷操作的概率分布函数的集合, 这些函数是时间 t 和工作负载 w 的函数.

扩充以后, 缺陷对程序的影响就被描述为程序中的一个操作, 这样就可以在相同的框架下讨论程序的出错行为. 由于缺陷操作的发生是随机的, 因此程序 Program$_F$的状态序列必须被赋予一个概率测度, 这已超出了本研究的讨论范围, 具体细节可以参考文献[18].

4 讨论

对于一个形式化定义的合理性, 需要从两方面进行讨论: 一是抽象性, 即形式化定义是否将一个概念定义在有着严格基础的数学对象上; 二是概括性, 即该定义是否能够概括现实世界中特定概念的不同现象. 具体到上面给出的形式化定义, 也需要进行这两方面讨论.

1) 基于状态的程序理论将程序的运行定义在函数(状态、状态迁移以及状态序列都是函数)这一数学对象上, 解释了程序运行的本质. 因此, 在该基础上定义的偏差、失效以及缺陷等概念也有着严格的数学基础, 能够支持进一步采用数学方

法对程序的行为进行分析和验证.

2) 现实世界中的程序偏差主要表现为程序功能的偏差、性能的偏差以及计算误差等方面, 这些偏差现象都可以通过状态偏差, 即定义 5 进行定义. 对于程序缺陷, 将其定义为不期望发生的操作集合以及每个操作发生的概率分布, 该定义能够概括来源不同的各种缺陷, 如软件内部缺陷、外部缺陷、瞬时缺陷以及持久性缺陷等.

5 程序容错能力的分级

首先给出容错程序的形式定义, 然后基于此讨论程序的容错能力.

用定义判断状态偏差并不现实, 因为程序的规格说明一般是抽象的、不可运行的, 因此无法找到正确状态进行比对. 但是通常情况下程序的规格说明给出了程序应当满足的性质, 因此可以通过检验状态序列是否满足这些性质来判断其是否发生偏离.

安全性和活性是程序行为的两类基本性质, 程序的任意一个性质都可以表示为一个安全性质和一个活性性质的合取[19]. 安全性质是程序的任何状态都不能违反的性质, 因此通常也称为不变式, 而活性是描述程序发展趋势的性质, 即对于行为的任何前缀, 都必须发展为某种趋势.

对于一个处于缺陷环境下的程序而言, 安全性通常是更为重要的性质. 虽然无法保证缺陷不会发生, 但必须保证缺陷激活后, 软件仍处于安全的状态. 为此, 有两个选择:

1) Fail-stop, 即停机失效, 当检测到软件处于不安全状态时, 立即停机. 这样软件始终处于安全状态, 但却破坏了活性的满足.

2) 暂时允许不安全状态的存在, 只要这些状态能够得到恢复, 使得后续状态能够继续满足安全性和活性.

显然, 这两种方法对于软件容错能力的要求是不同的. 为了从理论上分析软件的错误处理能力, 首先给出程序容错的形式定义, 然后据此展开讨论.

定义 9(程序的容错) 对于程序 $\mathrm{Program}_F = \langle \mathrm{PROGVAR}, \mathrm{ACTIONS}, \mathrm{ACTIONS}_F, F(t, w)\rangle$, S 是 $\mathrm{Program}_F$ 应当满足的安全性. $\mathrm{Program}_F$ 能够容忍缺陷 $\mathrm{ACTIONS}_F$ 并满足 S, 当且仅当存在一个谓词 T 满足以下条件:

1) 松弛性: $S \Rightarrow T$, 即满足谓词 S 的状态一定满足谓词 T, 或者 S 的特征集是 T 的特征集的子集;

2) 封闭性: 谓词 T 针对程序的正常操作和缺陷操作是封闭的, 即如果 $s_i \xrightarrow{\mathrm{act}} s_{i+1}$ 是一个迁移步, 而且 s_i 满足 T, 则 s_{i+1} 一定也满足 T. 或者说从 T 的特征集中的任意状态出发, 执行 $\mathrm{act} \in \mathrm{ACTIONS} \cup \mathrm{ACTIONS}_F$ 操作后, 结果状态仍然在 T 的特征集中;

3) 收敛性: 谓词 T 收敛到谓词 S, 即对于任意一个满足 T 的状态 s_i, 存在一个仅执行 ACTIONS 中的操作的有限状态序列 $s_i \xrightarrow{\mathrm{act}_1} s_{i+1} \xrightarrow{\mathrm{act}_2} s_{i+2} \rightarrow \cdots \xrightarrow{\mathrm{act}_n} s_{i+n}$, $\mathrm{act}_i \in \mathrm{ACTIONS}$, 使得 s_{i+n} 满足 S.

对以上定义做如下理解: S 是程序必须保持的安全性质. 缺陷 $\mathrm{ACTIONS}_F$ 的激活可能会使状态偏离期望的状态, 不再满足 S. 如果状态偏离局限在 T 所限定的状态内, 而且通过执行有限个正常操作能够把状态从 T 恢复到 S, 则程序就能够继续保持其安全性得到满足. 可见, T 是被用来刻画状态偏差程度, 即缺陷对系统状态的扰动程度的谓词, 被称为缺陷影响域(图 3).

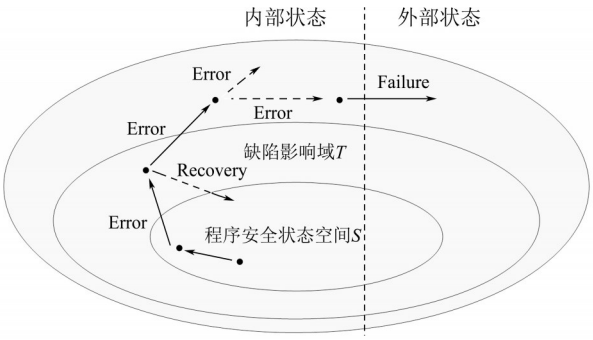

图 3 程序的容错

Fig. 3 Fault-tolerance for a program

根据以上定义对程序的容错能力进行如下讨论.

从两个维度对程序的容错能力进行分级: 稳定能力和屏蔽能力. 稳定能力是指程序允许状态发生偏差的程度, 以及偏差后能否恢复到正常状态的能力; 屏蔽能力是指程序不让用户觉察到状态发生偏差的能力.

对以上两个维度进行组合, 可以形成一个层次结构, 作为衡量程序容错能力强弱的度量(图 4).

位于顶层的“全局稳定+完全屏蔽”是最“完美的”容错程序, 能够屏蔽任何状态扰动(即状态偏差), 这是最理想的情形, 也是实现成本最高的情形; “局部稳定+完全屏蔽”和“全局稳定+降级模式”这是次好的情形, “局部稳定+降级模式”是最

表1 程序的容错能力的分级

Table 1 Hierarchy of fault-tolerance

	分级	条件	解释
稳定能力	全局稳定 (global stabilizing)	T=True ∧ T收敛	对于任意的状态偏差, 程序都能够恢复, 即程序能够容忍任意缺陷类型. 这种程序也称为“自稳定”程序[20], 其实现的成本是最高的.
	局部稳定 (local stabilizing)	T!=True∧ T收敛	只能对T所限定的状态偏差进行恢复, 不满足T的状态不能被恢复.
屏蔽能力	完全屏蔽 (masking)	T=S	缺陷操作执行的结果仍然保持其安全性, 也就是用户完全觉察不到缺陷的存在, 即程序能够完全屏蔽状态偏差.
	降级模式 (graceful degradation)	T!=S ∧ T收敛	用户能够观察到状态发生偏差, 但是修复操作能够最终使程序收敛到安全状态, 即在状态发生偏差期间, 程序不满足安全性, 但是程序仍然可以运行, 并最终恢复到安全状态.
	非屏蔽 （nonmasking）	T!=S ∧ T不收敛	用户能够观察到状态发生偏差, 而且出现偏差的状态不能得到恢复, 始终处于非安全状态. 这种情况下, 尽管安全性不能得到保证, 但程序至少还可以运行, 因此也具有一定的实际意义.
	无法预知	不存在满足定义中三个条件的T	程序对错误不具有任何处理能力, 这是最差的情形.

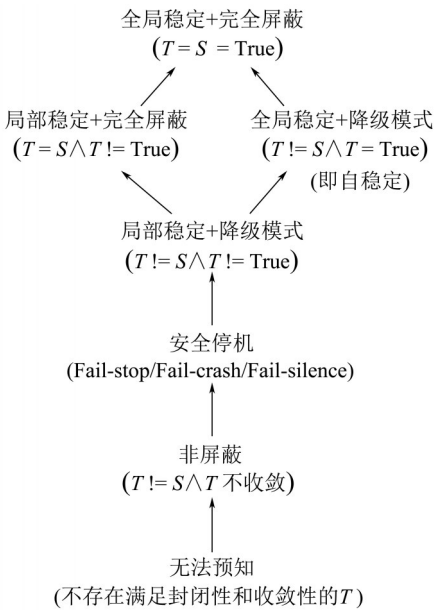

图 4 程序容错能力分级

Fig. 4 Hierarchy of fault-tolerance

常见的情况. “安全停机”要好于“非屏蔽”, 因为前者能够保持安全性, 而后者尽管可以运行, 但不能保证运行状态是安全的, 因此差于前者.

6 总结

本研究试图从程序的基本行为模型出发, 解释程序错误相关的基本概念, 并对程序的容错能力进行分级. 形式化的定义有利于形成对概念的准确理解; 有利于分析程序失效机制、探究程序错误处理方法; 能力分级有利于从理论的高度对程序错误处理能力的把握, 并对现有方法的适用性有清楚的认识.

本研究的最终目标是将程序看作一个“生命体”, 建立能够接受错误、容忍错误、愈合错误、适应复杂环境的程序, 并顺利完成其使命的理论框架. 显然, 本研究只是刚刚起步, 进一步的工作包括: 增加错误的概率模型、挖掘程序失效模式、探讨失效机制等.

参考文献

[1] Avizienis A, Laprie J C, Randell B, et al. Basic concepts and taxonomy of dependable and secure computing[J]. IEEE Transactions on Dependable and Secure Computing, 2004, 1(1): 11-33.

[2] Lyu M R. Software fault tolerance[M]. New York: John Wiley & Sons, 1995:10-35.

[3] Cristian F. A rigorous appro1ach to fault- tolerant programming[J]. IEEE Transactions on Software Engineering, 1985, 11(1): 23-31.

[4] Liu Z, Joseph M. Specification and verification of fault-tolerance, timing and scheduling[J]. ACM Transactions on Programming Languages and Systems, 1999, 21(1): 46-89.

[5] Bernardeschi C, Fantechi A, Simoncini L. Formally verifying fault tolerant system designs[J]. The Computer Journal, 2000, 43(3): 191-205.

[6] Reese J. Software deviation analysis[D]. Irvine: University of California, 1996.

[7] Trivedi K S, Vaidyanathan K, Goseva-Popstojanova K. Modeling and analysis of software aging and rejuvenation[C]//Simulation Symposium, 2000 Proceedings. 33rd

Annual, New York: IEEE, 2000: 270-279.

[8] Grottke M, Matias R, Trivedi K S. The fundamentals of software aging[C]//IEEE International Conference on Software Reliability Engineering Workshops, New York: IEEE, 2008: 1-6.

[9] Shelton C P, Koopman P, Nace W. A framework for scalable analysis and design of system-wide graceful degradation in distributed embedded systems[C]//Proceedings of the 8th International Workshop on Object-Oriented Real-Time Dependable Systems, New York: IEEE, 2003: 156-163.

[10] Dubey A, Mahadevan N, Karsai G. A deliberative reasoner for model-based software health management[C]// The 8th International Conference on Autonomic and Autonomous Systems. New York: ACM, 2012: 86-92.

[11] Psaier H, Dustdar S. A survey on self-healing systems: approaches and systems[J]. Computing, 2011, 91(1): 43-73.

[12] Kephart J O, Chess D M. The vision of autonomic computing[J]. Computer, 2003, 36(1): 41-50.

[13] 周明辉, 郭长国. 基于大数据的软件工程新思维[J]. 中国计算机学会通讯, 2014, 10(3): 37-42.

[14] 李宁, 李战怀. 软件缺陷数据处理研究综述[J]. 计算机科学, 2009, 36(8): 21-25.

[15] 赵廷弟. 安全性设计分析与验证[M]. 北京: 国防工业出版社, 2011.

[16] Dijkstra E W. Guarded commands, nondeterminacy, and formal derivation of programs[M]. New York: Springer-Verlag, 1975: 453-457.

[17] Kurki-Suonio R. A practical theory of reactive systems [M]. New York: Springer-Verlag, 2005: 32-45.

[18] Kwiatkowska M, Norman G, Parker D. Stochastic model checking[M]. Berlin Heidelberg: Springer- Verlag, 2007: 220-270.

[19] Schneider F B. On concurrent programming[M]. Berlin Heidelberg: Springer-Verlag, 1997: 42-48.

[20] Schneider M. Self- stabilization[J]. ACM Computing Surveys, 1993, 25(1): 45-67.

(责任编辑: 张 勇)