可信与安全计算领域的基本概念和分类

Algirdas Avi zienis, Fellow, IEEE, Jean-Claude Laprie,

Brian Randell, and Carl Landwehr, Senior Member, IEEE

**摘要**：本文给出了可信性相关的定义，包括可信性的特征例如可靠性，可用性，安全性（功能安全safety），完整性，可维护性等。安全性（系统安全）除了可用性和完整性之外还包含机密性。首先给出基本定义并进行评论后给出附加的定义，这些定义包括可信性和安全性（Security）的威胁是什么---故障，错误，失效，以及消除这些威胁的方法---故障预防，故障容错，故障排除，故障预测。

**关键词**：可信性，安全性（Security），信任，故障，错误，失效，漏洞，攻击，故障容错，故障排除，故障预测

# 1 简介

这篇论文给出可信和安全计算领域的各种概念的精确的定义。当一个系统的边界不确定的时候来定义这些概念是非常困难的，而且系统的复杂性也是主要的障碍。进一步来说，高复杂度的系统通常是主要的障碍，决定一个失效的可能原因或者后果将会是一个精细的过程，在避免产生失效时也会出现错误的故障预防方式。

可信性首次作为通用概念引入，其特征包含**可靠性（Reliability），可用性(Availability)，功能安全性（Safety），完整性（Integrity），可维护性（Maintainability）**等。安全性（Security）除了可用性和完整性之外还包含**机密性（Confidentiality）**。附加的定义会作为基础的定义的评论或者补充。使用粗体字（Boldface）被用来作为术语定义，斜体字（*italic*）主要用来吸引读者的注意力

本文可被视为达成各个专业领域对可信性概念的基本共识以促进卓有成效的技术互动的一种尝试。本文适合的读者1）标准化组织 2）教学用途 3）工程技术人员。

本文没有虚夸是最先进的资料，因此和我们聚焦的概念一起，我们并不解释标准中已经给出的执行措施层面的问题，例如在[30]中的功能安全（safety）和[32]中的系统安全（Security）。

可信性和安全性社区有如下的区分，总结为1）可信性社区认为针对非恶意性故障的限制仅仅解决了问题的一部分2）安全性社区认为需要从过去聚焦在机密性（confidentiality）上面扩大到完整性（integrity）和可用性（availability）上面，虽然已经放在security的定义里面了，但是目前仍然没有引起重视。尽管本文目的在于提供可信性和安全性放在一起的共同方面，但是限于篇幅原因，机密性仍然没有详细的描述。

**以前的工作和未来的目标（Preceding Work and Goals for the Future）。**这些技术研究的源头可以回到1980年，当时一个叫做“概念和术语基础“（Fundamental Concepts and Terminology）的联合委员会被TC成立在Fault-Tolerant Computing of the IEEE CS and the IFIP WG 10.4 “Dependable Computing and Fault Tolerance”。几篇有价值的的论文发布在1982年的FTCS-12 [21]，在1985年的FTCS-15 [40]做了整合，并成为了本文的前身，但是当时在没有提供太多的详细分类，特别是在可信性的威胁和特征上面。

关于可信性的持续的强化讨论导致在1992年出版了《*可信性：基础概念和术语*》一书*《Dependability: Basic Concepts and Terminology》* [41]，这本书包含了34页的英文正文内容以及8页的词汇表，并且被翻译为法文，意大利文，德文和日文。这本书的重要的创新点是把*系统安全（Security）*作为一个特征（attribute）加入*，故意的恶意故障（intentional malicious faults）*作为故障（Faults）的术语加入，很多概念被重新定义和详细描述了。

接下来的主要步骤是认可安全性（security）是由*机密性、完整性、可用性（confidentiality, integrity, and availability）*这几个特征组合而成的，并且给出了*故意的非恶意故障（intentional nonmalicious faults）*的分类，以及*不充分的系统规格问题的分析*[42]*（analysis of the problems of inade -quate system specifications），*尽管这被视为仅仅提供了可信性威胁的一个总结性的分类。

本文表述了自从1995年以来的所有的持续努力的成果，这些努力包括扩展、提炼、简化可信性和安全性（security）的分类。我们努力提供针对于工程实践和学术领域都容易读懂的分类，因此本文也包含了上面提到的出版物中的内容。主要的新贡献如下：

* *可信性和安全性的关联关系*（2.3节）*The relationship between dependability and security* is clarified (Section 2.3).
* *可信性的定量的定义（2.3节）A quantitative definition of dependability* is introduced(Section 2.3).
* 人为的非恶意故障的*分类标准（3.2.1和3.2.2节）The criterion of capability* is introduced in the classification of human-made nonmalicious faults (Sections 3.2.1 and 3.2.3), enabling the consideration of competence
* *恶意故障讨论的更新*（3.2.4节）*The discussion of malicious faults* is extensively updated (Section 3.2.4).
* 将*业务失效*（3.3.1）和*可信性失效*（3.3.3）区分开来，后者指的是频繁的、严重的持续一段时间的业务失效*Service failures* (Section 3.3.1) are distinguished from *dependability failures* (Section 3.3.3): The latter are recognized when service failures over a period of time are too frequent or too severe.
* *开发过程中的可信性*明确进入分类，包含部分和整体的开发过程失效（3.3.2节）*Dependability issues of the development process* are explicitly incorporated into the taxonomy, including partial and complete development failures (Section 3.3.2).
* *可信性相关的可依赖性和可信任性概念*（4.2节）以及对比了可信任性、可生存性、高自信性三个相同的概念（4.4节）The concept of *dependability is related to dependence and trust* (Section 4.2), and compared with three recently introduced similar concepts, including survivability, trustworthiness, high-confidence systems (Section 4.4).

在当前的扩展迭代之后，我们可预见的关于这个分类的进化与提升的将来的机会和挑战是什么呢，非常确定的是，我们可以得出如下的将来的愿望：

* 展开讨论安全性：expanding the discussion of security，例如保障机密性的关键技术，建立确定性（establishing authenticity）的技术等等
* 分析可信任（trust）以及展开风险管理的话题
* 寻找可信性和安全性的统一度量方法

我们希望一些挑战将会超出我们的预期（也许叫做所谓的紧急属性“emergent properties”例如the HAL computer in Arthur C. Clarke’s “2001: A Space Odyssey”），因为我们构建了超出我们的理解能力的复杂的人机系统。其他的挑战也比较容易预测：

* 新技术（nanosystems, biochips, chemical and quantum computing）和新概念的人机系统（ambient computing, nomadic computing, grid computing）将会持续要求关注他们的可信性
* 复杂的人机交互问题持续成为一个日益严重的挑战。提升可信性和安全性的方法需要被找打并且包含进来
* 人性的另一面（黑暗）使得我们可以预料到一些新型的恶意从而产生较多的恶意故障，因而也需要新的保护措施

从上述的挑战的视角来看，并且因为持续的不必要的难以理解的据称的“新”的概念介绍来描述相同的方法、特征、威胁。在将来最重要的目标是保持这些分类最大可能的完整性，但同时是最简单的而且在我们的能力范围内。

# 2 基本概念

在本节，我们提供了贯穿整个可信性和安全性计算领域分类的全部讨论的一系列基础定义。这些定义充分覆盖了计算和通信领域的全部范围，从单个的逻辑门到人类操作和使用的网络计算机系统。本文我们将聚焦在计算和通信系统，但是我们的定义对于相关的**基于计算机的系统（computer-based systems）**

## 2.1 系统功能、行为、架构、服务

**系统（System）**在本文中定义为一个和其他的实体（entity）交互的实体，其他系统包括硬件，软件，人，物理环境。其他系统是某个系统的**环境（Environment）**。因此**系统边界（**system boundary）指的是系统和其所处的环境的交界处（frontier）。

计算和通信系统的基础特性（Fundamental Properties）包括*功能（Function），性能（Performance），可信性（Dependability），安全性（Security），成本（Cost）*。其他影响可信性和安全性（Security）的重要特性包括*易用性（Usability），可管理性（Manageability），自适应性（Adaptability）*等，对这几个特性的详述超出了本文的范围。

系统**功能（system function）**指的是系统能够做什么，通常通过**功能规格（functional specification）**来定义。系统**行为（behavior）**是通过功能执行来体现的，被描述为一系列的系统状态。系统**总的状态（total state）**表现为计算、通信、存储信息、互联、健康检测等。

系统**架构（Structure）**指的是实现系统行为的方式，基于结构化视角来看，系统是由一系列的**组件（components）**组成的，组件之间有彼此的边界，每个组件都是一个系统（System）。组件是可以逐层往下分解的，直到分解到是**原子的（atomic**，不可分解的或者再分解已经没有意义了）。所以说，一个系统的总状态（Total State）是由系统内部所有原子组件的所有外部状态构成的。

系统（角色为**provider**，名称为A）提供的**服务（service）**，指的是被他的用户感知到的系统行为。**用户（user）**指的是接受服务系统（Provider）提供服务的另一个系统。系统（Provider）提供服务的边界就叫做**服务接口（Service Interface）。**在服务接口可感知的状态叫做系统的外**部状态（external state）**，其余的状态称之为**内部状态（internal state）**。一个系统可能同时的或者先后成为另一个系统的提供者（Provider）或者用户（User），向另一个系统提供服务或者接受另一个系统的服务。用户接受服务的界面被称之为**用户接口（user interface）**

我们到目前为止讨论的都是单个功能（Function）和服务，但是实际上一个系统是包含多个功能（Functions）和提供多个服务的。单一的功能和服务可以看作是功能目录和服务目录的组成部分。为了简化的目的，我们使用复数形式的功能（functions）、服务（services）当我们需要区分多个功能（function）和服务（service）目录的时候。

## 2.2 可信和安全的威胁：失效（Failure）、错误（Error）、故障（Fault）

当系统功能被正确执行时说明系统提供了正常的业务（correct service）。**业务失效（service failure）**也通常简称为**失效（failure）**，定义为当系统的业务偏离正常的业务时的一个事件（event）。失效的原因可以是没有按照功能说明书来实现也可是功能说明书没有充分体现功能需求。

业务失效是一个系统从正常业务到非正常业务（Incorrect Service）的一个**转折点（transition）**,非正常业务的例子如没有实现系统功能。非正常业务持续期间就叫做**业务中断（service outage）**。从非正常业务到正常业务的转折点（Transition）就叫做**业务恢复（service restoration）**。偏离正常业务的具体表现形式称为**失效模式（service failure modes）**，通过**失效严重程度（failure severities）**来体现不同的失效等级。下面的章节会有失效模式的详细分类。

既然一个业务是系统的外部状态的结果体现，因此当业务发生失效时一定是一个或者多个系统的外部状态偏离了正常业务状态。这种偏离定义为*错误（error）*。引发错误的明确的或者潜在原因称之为**故障（fault）。**故障产生于系统的内部或者是系统的外部。由于系统会存在**漏洞（vulnerability**，例如一个内部故障使得外部故障来危害系统），因此会出现一个外部故障引发系统内部的错误（Error）,进而还可能会导致业务失效（Failure）。大部分情况下，一个故障首先会引发一个系统组件内部错误（系统内部状态的一部分），系统外部状态是没有受到影响的。

基于上述原因，**错误（error）**也被定义为系统所有状态中可能会导致后续业务失效的那个状态，需要重点说明的是很多的错误并不会引发业务失效。故障被激活（active）的时候才会产生错误（error），否则故障就是**休眠的/静默的（dormant）**

当某系统的功能规格包含多个功能的时候，一个或者多个业务失效会引发系统出于**降级（degraded mode）**工作状态（一部分业务仍在服务用户），系统降低可以表达为如下的这几种模式：业务降速（slow service）,业务受限（limited service）, 紧急模式（emergency service）等等，我们也可以说系统是处于功能或者性能的**部分失效（partial failure）**状态。

## 2.3 可信和安全（Security）的特征（Attributes）

**可信性**（Dependability）的原始定义是系统提供业务的能力是能够让人充分信任（trust）的，这个定义的重点是充分信任（justification of trust）**。可信性**的另一个定义是提供一个标准，通过此标准可以判断系统是否能够避免出现高频的和严重的不可接受的业务失效。

通常来说一个系统的可信性能够让另一个系统可以依赖，系统A对系统B的**依赖（dependence）**可以表达为系统B会影响系统A的可信性。依赖的概念导致**信任（trust）**这个词可以方便的定义为*可接受的依赖（accepted dependence）。*

根据前面的的讨论，可信性是一个综合的概念，包含如下的几个特征：

* **可用性：Availability**，随时提供正常业务的能力
* **可靠性：Reliability**，持续提供正常业务的能力
* **安全性：Safety**，人和环境免于灾难性后果的能力
* **完整性：Integrity**，没有不当的变更
* **可维护性：Maintainability**，对系统进行修改和维修的能力

当谈到安全性（security），一个附加的特征显得特别突出，那就是**机密性（confidentiality）**，其含义为不存在非授权的信息访问。**安全性（Security）**也是由机密性（confidentiality）, 完整性（integrity）和可用性（availability）等特征组成的，并且需要同时满足1）授权（authorized）下的可用性 2）机密性confidentiality）3）未经授权下的完整性（integrity with improper meaning unauthorized）。下图Fig1总结了可信性和安全性的基本特征：这幅图*不*应该被解释为表面的那样，例如安全开发者对可维护性不感兴趣或者在可信性领域不研究机密性，而应该是在兴趣和活动之间取得最大的平衡。

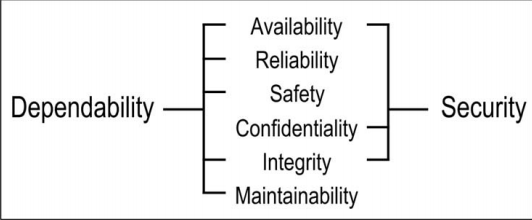


Fig1

一个系统的可信性和安全性规格（dependability and security specification）必须要满足在特定类型的故障和指定用户环境下所能够接受的业务失效的频率和严重程度。对于特定的系统来说，一个或者多个特征不是同时适合的

## 2.4 构建可信性和安全性的方法

在过去的50年里出现了很多构建可信性和安全性（Security）的方法，这些方法可以分为4大类：

* **故障预防：Fault Prevention**，预防故障的发生和传播
* **故障容错：Fault Tolerance**，在发生故障的时候避免出现业务失效
* **故障清除：Fault Removal**，减少故障发生数量，降低故障严重程度
* **故障预测：Fault Forecasting**，预测未来故障发生的可能性

故障预防和故障容错目标在于能够提供可以信任（trust）的业务，故障清除和故障预测在于通过评估系统的功能、可信性、安全性规格的充分性以及当前的满足度来获得对系统的信心。可信性和安全性的全景图如下Fig2：

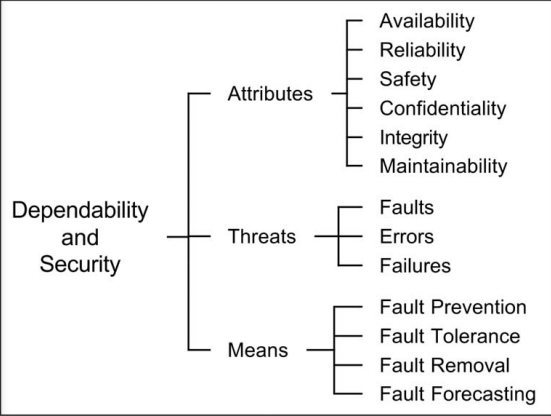


Fig2

# 3 可信性和安全性的威胁

## 3.1系统生命周期

在本节会介绍系统全**生命周期内（life cycle）**的威胁。一个系统的生命周期包含两个阶段：*开发阶段和使用阶段(development and use)。*

**开发阶段（development phase）**包含从客户原始需求到系统充分测试并达到用户环境使用标准的一系列的活动过程。在这个开发阶段，系统会和开发环境产生交互，从而会将开发缺陷（development faults）引入到系统中。一个系统的**开发环境（development environment）**包含如下的要素：

* *物理环境：physical world*， 物理世界的自然现象
* *开发者：human developers，*一些人可能缺乏能力，也有一些人存在恶意行为
* *开发工具：development tools，*协助开发的软件或者硬件
* *制造和测试设施：production and test facilities*

**系统的使用阶段**（The use phase）开始于当这个系统已经准备好并且可以让用户使用业务的时候。业务使用包含**正常业务的提供（service delivery）**，**业务中断（service outage）**，**业务关闭（service shutdown）**。业务中断是由业务失效引发的，具体体现为在业务界面没有业务或者提供不正常的业务，业务关闭指的是系统被授权实体主动挂起。**可维护性活动**会发生在使用阶段的三个时期。

在使用阶段，系统会和它的**使用环境（use environment）**进行交互，不可避免的会受到使用环境中产生的故障的影响。**使用环境**包含如下的要素：

* *物理环境：physical world*， 物理世界的自然现象
* *管理员：administrators*，包含维护人员，一些人可能缺乏能力，也有一些人存在恶意行为
* *用户：Users*，使用系统业务的实体（人或者其他系统）
* *业务系统：Providers*，提供业务的实体
* *基础设施：Infrastructure*，为业务系统提供特定服务的实体，例如Time、定位服务、供电系统，散热冷却系统等等
* *入侵者：Intruders*，企图越权访问，变更系统业务的实体，包含黑客、破坏者、内部腐败人士，敌对人员、恶意软件等等。

**可维护性（maintenance）**这个词在这里遵循通用的定义，不但包含系统的维修（repair），也包含在系统使用期间的对象提供的变更/修改操作。因此可维护性其实是一个开发过程（development process），后续关于开发过程的讨论都适用于可维护性，全景图总结如下图Fig3所示

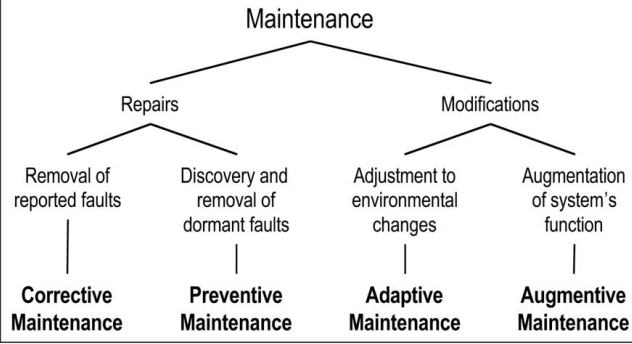


Fig3

值得我们注意的是维修（repair）和故障容错（fault tolerance）的概念具有一定的相似性，在本文中故障容错和可维护性的区别在于可维护性有系统之外的外部代理（维修者repairman，测试装备，远程重加载软件等）的参与，进一步来说维修是在是系统使用阶段的故障清除（fault removal）,故障预测（fault forecasting）也可以认为是一种维修（repair）的情况。实际上假如我们把被维修系统，使用人/维修人以及实施维修的系统放在一起作为一个大系统的话，维修可以认为是一种故障容错活动。

## 3.2故障Fault

### 3.2.1 *故障的分类（Taxonomy of Faults）*

所有在系统生命周期当中影响系统运行的故障（Fault）可以从8个视角进行分类，得到了如下的*基础故障类别（elementary fault）*如下图Fig4所示：

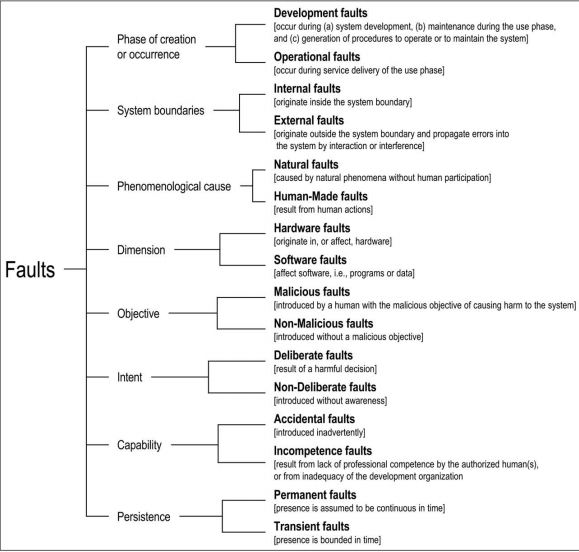
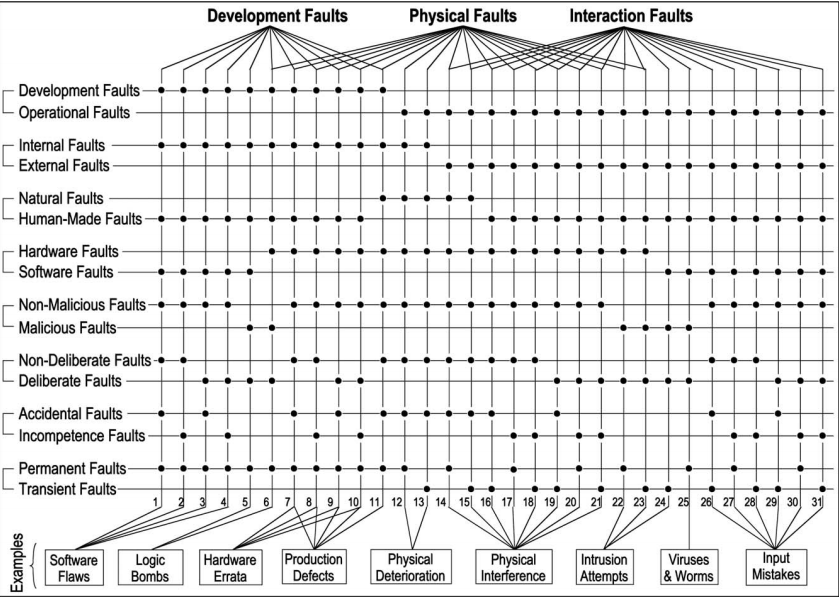


Fig4

如果这8个基础分类的故障可以组合在一起的话，我们会得到256种不同的*组合故障（combined fault classes）*。然而并非所有的标准都适用于所有的故障，因此我们总结出*31种可能的故障组合（likely combinations）*，参见Fig5

在将来我们可能会有更多的组合，在Fig5展示了三种部分重叠分组的组合故障类型：

* **开发故障**：development faults，在开发阶段产生的所有故障
* **物理故障**：physical faults，所有的硬件故障
* **交互故障**：interaction faults ，所有的来自于外部的故障



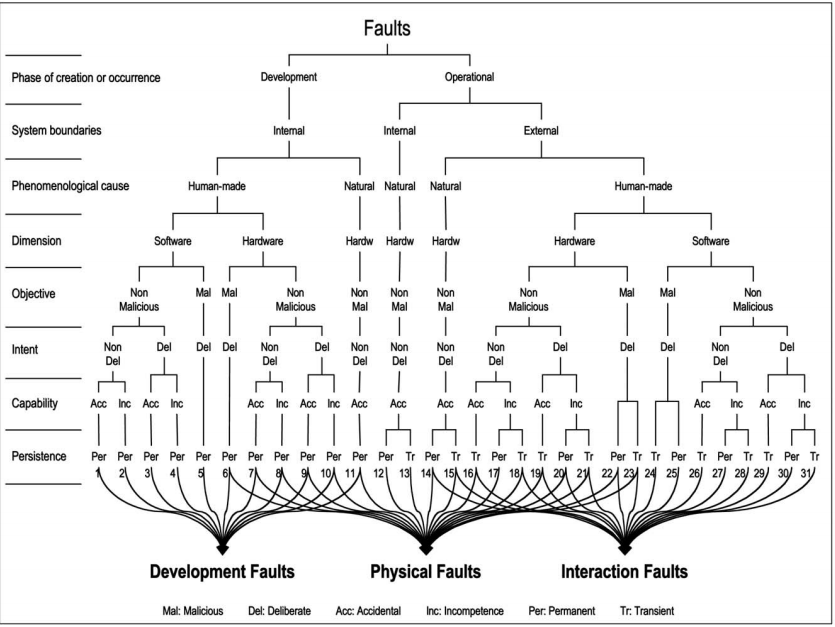


Fig5

列举出所有的可能故障类型可以让用户决定哪些故障类型需要纳入到可信性和安全性的规格中。接下来我们继续讨论Fig5中的故障类型：

### 3.2.2 自然故障（Natural Faults）

自然故障（11-15）指的是物理（硬件）故障（自然而然产生的非人为参与的）。不过呢，人也可以引发物理故障（6-10,16-23），在下面将会讨论。*生产缺陷（Production defects）*是产生于开发过程的一种自然故障。自然故障要么是因为自然过程引发的物理恶化而产生的*内部故障*（12-13）、要么是因为硬件边界的自然穿透（例如辐射）或者是通过用户接口（例如供电）引入的外部故障（14-15）

### 3.2.3 人因故障（Human-Made Faults）

人因故障指的是由于人的行动而产生的故障，包括应该执行的行动没有执行，例如**遗漏性故障（omission faults）**，也简称为**omissions**。执行错误的行动导致的故障称之为**过失性故障（commission faults）**。

以下两种基础的人因故障的区别主要是开发者或者使用期间的人与系统的交互的客观性。

* *恶意故障：Malicious faults*，开发阶段引入的危害使用阶段的故障（5-6）或者是使用阶段直接引入的故障（22-25）
* *非恶意故障*：*Nonmalicious faults（*1-4，7-21，26-31），非恶意引入的故障

我们首先考虑非恶意的故障，可以从开发者的意图的角度进行划分：

* *客观性故障：nondeliberate faults*，是一种客观失误（mistake），是由于开发者，操作者，维护者的无意的/无心的动作（unintended actions）而产生的 (1, 2, 7, 8, 16-18, 26-28)
* 过失性故障：deliberate faults，是一种*错误决定（bad decisions）*，是由于*非预期（intended actions）的错误行为*而导致的(3, 4，9, 10, 19-21, 29-31)。

过失性，非恶意的开发故障(3, 4，9, 10)来自于两种权衡和取舍，1）在提高新的利用率的时候维持可接受的性能 2）经济因素的考虑。过失性，非恶意的开发故障交互故障（19-21, 29-31)主要来自于操作者的不当的行为，一是未克服不可预见的场景，二是因为没有意思到潜在的后果而故障违反操作程序。过失性，非恶意的故障通常被认为是一种不可接受的系统行为，失效就接着发生了。开发者或者操作者在那个时刻并没有意识到他们的这个决定的后果是一个故障。

通常认为不管是失误（mistake）还是错误决定（bad decision）都是一种*意外（accidental）*，主要是他们不是因为恶意的做法引发的。然而*并非所有的（not all）*非恶意的人产生的所有的失误和错误决定都是意外。一些非常有危害的失误（mistake）和非常糟糕的决策（bad decision）都是由于缺乏对这个任务的专业能力所导致的，一个完整的故障分类应该包含这类故障。因此我们进一步区分非恶意的人因故障为1）*意外性故障（accidental faults）*和2）*能力不足故障（incompetence faults）*，具体的分类参见下图Fig6：

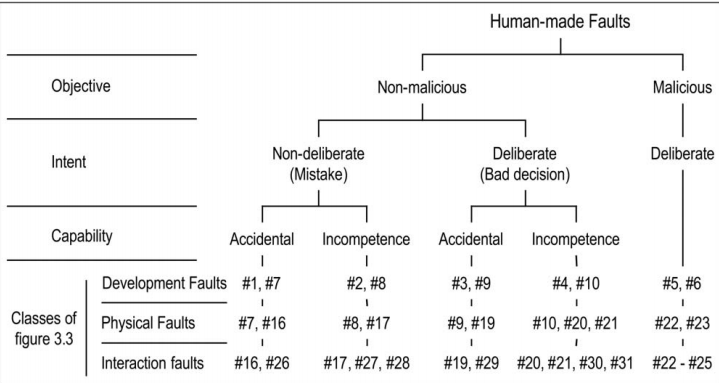


Fig6

如何识别能力不足故障在失误（Mistake）和错误决定（Bad Decision）导致了较大的经济损失、重大伤害、人员伤亡的情况下会得非常重要。在这种情况下，类似于法庭的合法调查过程或者咨询委员会的那种独立专业评估用来进行故障类型判断。

到目前为止关于能力不足故障的讨论还是独立的，通常人为的努力会因为团队/组织的组织能力不足而失败。组织能力不足的典型的例子就是AAS系统的开发失败，该系统是为了代替老的飞行控制系统[67]。

非恶意的开发故障在硬件和软件里面都会存在，硬件领域尤其是微处理器，在产品应用之后也发现了较多的开发故障[5]，这样的故障通常叫做勘误表（“errata”），会在后续的更新列表里面体现。在处理器的整个生命周期里面都会产生errata，因此更新列表会周期性发布。我们使用的开发工具也会产生开发故障。

系统设计中不可避免的要使用现成的（Off-the-shelf (OTS)）组件，这会带来额外的问题，这些问题可以是已知的开发故障也可能是未知的，而且它的规格是不完整的甚至是错误的。这些问题在遗留的（Legacy）OTS组件被使用的时候会尤其严重，因为这些*Legacy* OTS是以前设计使用的但是由于用户需求的原因不得不保留下来。

一些软件的开发故障可能会导致软件老化（software aging），逐渐累积的错误会导致性能下降甚至完全失效。例如内存泄漏、未结束的线程、未释放的文件锁、数据损坏、存储碎片、四舍五入（round-off errors）的累积效应。

### 3.2.4 恶意故障（Malicious Faults）

*恶意的人因故障（Malicious human-made faults）*指的是在使用过程中恶意破坏系统功能而产生的结果。这些故障的目标是1）扰乱或者挂起服务，引发**拒绝服务denials of service** 2）获取机密信息 3）非法修改系统。分为两种类型：

* **恶意逻辑故障（Malicious logic faults）**：包括类似于*Trojan horses*，logic or timing *bombs,* *trapdoors*这类的开发故障和类似于*viruses, worms, zombies*这样的操作类故障[25]（operational faults）在[39][55]中定义的这类故障参见Fig7
* **入侵类故障（Intrusion attempts）**：是一种外部操作类故障，也包括是由于超出了自身权限的系统操作员或者系统管理员所导致的。一般是采用物理的手段来触发这类故障，比如供电中断，辐射，窃听（wire-tapping），高温，低温等等。

我们所俗称的漏洞利用（exploit）就是一个存在漏洞（vulnerability）的系统脚本，这个脚本可以让入侵者能够访问甚至于控制整个系统。在这里exploit定义为操作类的，外部的，人因的，软件的、恶意的，交互的故障（24-25）。使用电吹风加热一个内存条（RAM）会触发内存错误，这种内存错误就会允许软件安全违法行为，这种行为就是一个外部的，人因的，硬件的、恶意的，交互的故障（22-23）。漏洞利用者（exploit）所利用的漏洞其实就是一个软件瑕疵（flaw，例如未检查的缓存），可以归类为开发过程的，内部的，人因的，软件的，非恶意的，非过失性的，永久性的故障（1-2）。

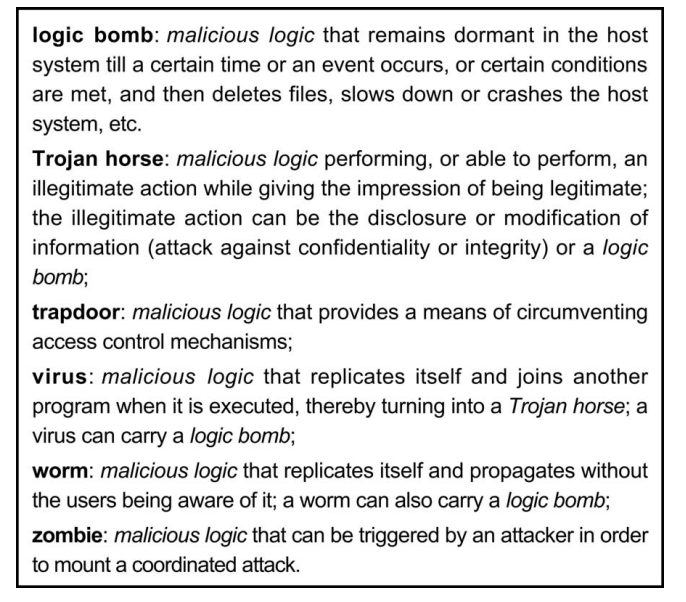


Fig7

### 3.2.5 交互式故障（Interaction Faults）

交互式故障发生在产品使用期间，因此他们也全是*操作类故障（operational faults）*，是使用环境的元素和系统的交互而引发的，因此也全部是*外部故障（external）*。这当中大多数是环境因素中的人的行为（human action）造成的，因此也是*人因故障（human-made）*，属于Fig5的16-31故障类别。这其中的一个特例就是由宇宙射线，太阳耀斑（cosmic rays, solar flares）引起的外部的自然故障（14-15），属于一种无人为参与的自然（nature）和系统的交互。

人因的操作类的故障最多的就是**配置故障（configuration faults）**，例如参数的错误配置会影响安全性（security），网络（networking），存储、中间件（storage, middleware）等[24]。这样的故障可以发生在配置改变期间的操作，这些配置变更操作是自适应的或者增强性的维护连同例如版本升级的系统操作。因此也叫作**重配置故障**[70] **(reconfiguration faults)。**

正如在2.2节提到的，交互类故障的共性特点是要成功触发一个交互式故障，那么在系统内部一定是有*漏洞的（vulnerability）*，外部故障正通过这个内部的故障来起作用的。这个内部故障可以是开发过程或者操作类故障，可以是恶意的和非恶意的故障，也可以是被漏洞利用的外部故障。入侵故障和缺乏保护的漏洞利用的物理的外部故障非常的相似。一个漏洞（vulnerability）来自于过失性的开发故障，这种故障是由于成本因素或者易用性原因，最终导致了有限的保护机制甚至保护机制缺失。

## 3.3失效Failure

### 3.3.1 业务失效

在2.2节，业务失效指的是当前正在运行的业务偏离了正确（Correct）业务的现象（Event），这种偏离（deviation）所表现出来的方式称之为*业务失效模式（service failure modes）*，每个失效模式都有不同的*失效严重等级（service failure severity）*。失效在第2章定义为系统当前的功能实现与系统功能规格说明书不符，但是如果规格中没有覆盖故障场景的话，尽管业务实现与规格相符，但是仍然是不满足客户的。这种规格类故障（specification faults）可以是遗漏性故障（omission faults），也可以是过失性（误解、错误的假设、不一致、印刷错误）故障（commission faults）。在这种情况下，不希望的事件（失效）只能在他发生后才能被发现（例如通过产生的后果）。因此失效可能是主观性或者存在争议的，需要一个评价组织进行识别和裁决。

业务失效模式从以下四个视角来体现不正确的业务：

* 失效域（the failure domain）
* 失效的可检测性（the detectability of failures）
* 失效的一致性（the consistency of failures）
* 失效的后果（the consequences of failures）

从**失效域（failure domain）**的角度我们区分失效类型如下：

* **内容失效**（content failures）：到达用户的业务内容错误
* **时效失效**（timing failure）：业务到达用户的时间错误或者业务的持续时间错误

内容和时间都不正确的失效可以分为如下几种类型：

* **挂起失效**（halt failure）：或者简称为挂起（halt），就是用户无法感知到系统的业务。一个特例就是**静默失效（silent failure）**，或者简称为**静默（silence）**，就是在业务接口处没有任何业务。
* **不稳定失效**（erratic failures）：非挂起失效(例如断断续续的，卡慢卡顿等等)

失效域视角的失效模式分类参见下图Fig8：

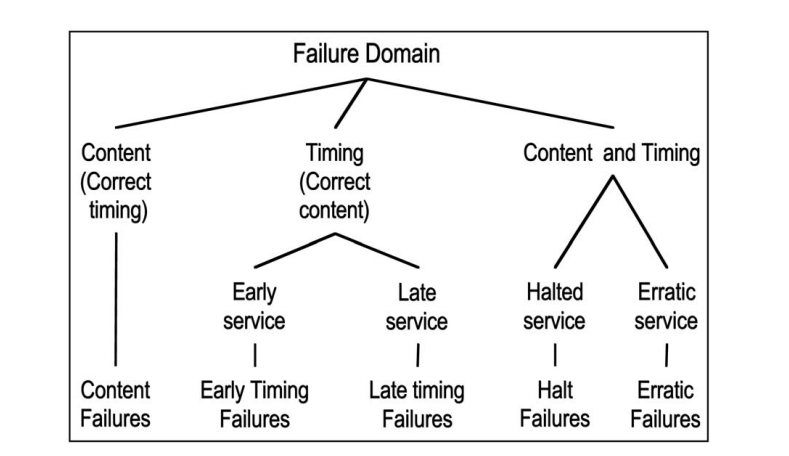


Fig8

**失效可检测（detectability）的视角**指的是客户可感受到的业务*失效信号（signaling）*。通过系统内部的检测机制能够检测到不正确的业务。当业务消失并产生一个告警信号的时候，则**信号失效（signaled failures）**发生了。检测机制本身有两种失效模式，1）误报，实际没有失效发生 2）没报，失效的时候没有信号报上来**（unsignaled failure）**。当业务失效的发生导致业务模式减少的时候，用户的业务处于一个降级状态（degraded modes），这种降级状态可以是从稍微减少到紧急业务状态甚至安全关闭。

**失效模式一致性的视角**指的是当一个系统有2个及以上的用户的时候，可以做如下的分类：

* **一致性失效（consistent failures）：**对于所有用户的失效模式都是一样的
* **非一致性失效**（**inconsistent failures**）：不同的用户有不同的失效模式（可能部分用户是未失效的）。也通常叫做拜占庭失效[38]**（Byzantine failures）**

失效严重程度指的是*失效后果（consequences of the failures）*的定级（Grading）。所有失效模式可以分为不同的严重等级（severity levels），等级通常和最大的可接受发生概率相关。严重等级的数字、标签、定义、可接受的发生概率都是应用（application）相关的，并且是包含到指定应用的可信和安全特征中的。具体的分级因素如下：

* 对于可用性：中断持续时间（the outage duration）
* 对于安全性（safety）：造成人员伤亡的概率
* 对于一致性（confidentiality）：发生暴露的信息的类型
* 对于完整性（integrity）：数据损坏的范围以及修复的可能性

通常来说，两种固定严重等级可以根据未有业务失效的好处以及失效后果的关联关系来进行定义：

* **轻微失效**（minor failures）：失效后果的危害性和获取不产生失效的代价一样的时候
* **灾难性失效**（catastrophic failures）：失效后果的危害性远远大于获取不产生失效的代价

下图Fgi9总结了各种失效模式：

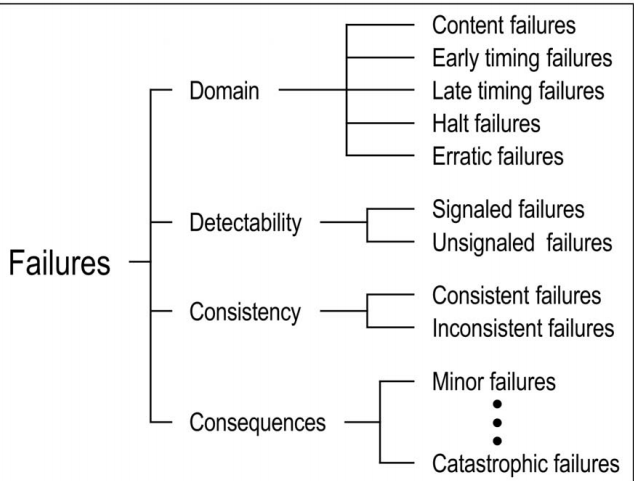


Fig9

被设计和实现成为仅会产生在可信和安全性规格里面定义的指定的失效模式的系统称之为**可控失效系统（fail-controlled systems）**，例如固定输出而不是不稳定的输出，静默而不是发出声音，一致性失效而不是非一致性失效；一个系统仅仅会产生挂起失效叫做**fail-halt (or fail-stop)**系统；仅有固定业务失效的系统叫做**fail-passive systems**；仅有静默失效的系统叫做**fail-silent systems**；仅产生轻微失效的系统叫做**fail-safe system**。

正如在第二章定义的那样，输出不正确的业务叫做*中断（outage）*，会一直持续到*业务恢复（service restoration）*。业务中断持续时间会因为业务恢复动作不同而显著的不同：1）自动或者自助式的恢复方式（recovery, restart, or reboot）2）修复性的维护。在业务恢复后的修复开发故障（通过补丁或者其他方式）通常是离线的，升级操作还需要选择合适的时间进行，升级过程也有可能是中断业务的。升级或者例行性维护导致的抢占式的系统中断称之为*计划性中断（planned outage）*

### 3.3.2 开发过程失效（Development Failures）

正如在3.1节定义的那样，开发故障是在系统开发的过程因为组织人员，开发工具，加工制造而引入的。这些开发故障可能造成部分的或者全部的开发过程失效或者直到网上才被发现。一个完全的开发过程失效可能会造成项目终止。有如下两个类型的开发过程失效：

* 预算失效（Budget Failure）：在项目开发完成之前预算已经用完
* 计划失效（Schedule failure）：在项目交付的时候技术已经过时或者功能无法完全满足用户需求

造成开发过程失效的主要原因有：不完整或者错误的规格、频繁的用户原始需求变更、不充分的功能/性能设计、太多的开发故障、不充分的故障清除（Fault Removal）能力、可信性和安全性的非充分预测、开发投入的错误估算。以上所有的原因都是因为低估了系统开发过程功能的复杂性。

有两种类型的部分开发过程失效（没有项目终结），第一种是当项目完成的时候，预算和计划都已经超支了；第二种是发布出去的系统功能少、性能低或者可信性和安全性未达到原始规格要求，叫做降级（downgrading）。

开发过程失效，无论是超时（Overruns）还是降级（downgrading），都会有在客户群体有很大负面影响。可以参考一些大型软件项目的数据统计或者直接看一下彻底失败的美国AAS项目（其损失达到1.5亿美元）。

### 3.3.3 可信和安全失效（Dependability and Security Failures）

可以预期的是在系统的使用期间会发生各种各样的故障影响系统的运行。这些故障可能会导致性能降级或者彻底的业务失效。基于这样的原因，可信心和安全性规格（dependability and security specification）是非常必要的，用于定义如下质量特征的质量目标：可用性、可靠性、安全性、一致性、完整性、可维护性（availability, reliability, safety, confidentiality, integrity, and maintainability）。

这个规格需要定义在用户使用环境（use environment）中会出现的各种故障，也需要给出某种不希望出现的或者危险情况下的保障措施，进一步来说，需要包含客户所需要的关于故障预防、故障容错的关键技术。

可信和安全性失效（dependability or security failure）发生在频繁的出现不可接受的系统业务失效。

可信和安全性规格也可能会有故障，一种是因为用户使用环境描述不完整或者需要进行预防/容错的故障类型选择不合理所造成的遗漏性故障（Omission faults）；还有一种就是对于一个或者多个特征高要求的不正确选择（质量特征规格要求过高），这种选择会导致开发费用超支或者开发过程失效。例如初始的AAS系统的可用性是业务中断3秒每年，后来被修改为5分钟每年。

## 3.4 错误（Errors）

正如在2.2节定义的那样，错误指的是会导致失效的所有系统状态的一部分，当一个错误引发了当前服务用户的业务偏离了正确的业务的时候就产生了失效。引起错误（Errors）的原因就叫做故障（Fault）

当一个错误信息或者错误信号（error message or error signal）出现的时候，那么这个错误就被检测到了（detected）。如果一个错误发生了但是没有被检测到就叫做静默错误（Latent Error）。

既然一个系统包含了多个互相联系的部件，所以所有组件的状态集合就是系统的左右状态。如上的定义也就意味着一个故障一开始在一个或者多个部件内产生一个错误，但是当这个部件的外部状态不是系统的外部状态的时候业务失效是不会发生的。每当一个错误成为一个部件的外部状态的时候，只是这个部件发生了业务失效，对整个系统来说这个错误仍然是内部的。

错误是不是会导致业务失效取决于如下两因素：

1. 系统架构，尤其是系统的冗余机制。

* 保护性冗余（protective redundancy）：提供一个故障容错机制来阻止错误引发的业务失效
* 无冗余机制（unintentional redundancy）：实际上建立一个没有任何冗余的系统即便有可能达到，那也是非常困难。

1. 系统行为：产生错误的部分不会提供服务或者错误在导致业务失效前被清除掉（removal）。

对于错误的分类可以采用3.3.1节中业务失效的方式进行分类（因为错误可以看做是一种初级失效--内部失效）：内容性错误或者时效性错误（content versus timing errors）, 可检测错误或者静默错误（detected versus latent errors）, 一致性错误和非一致性错误（consistent versus inconsistent errors）,微小错误和灾难性错误（ minor versus catastrophic errors）。

一些故障（例如电磁辐射）可能会同时在多个部件触发错误，这些错误叫做多关联性错误（multiple related errors），仅影响一个部件的错误叫做单一错误（Single errors）。

## 3.5 失效路径（Pathology of Failure）

失效、故障、错误的产生和传播机制参见图Fig10.具体解释如下：

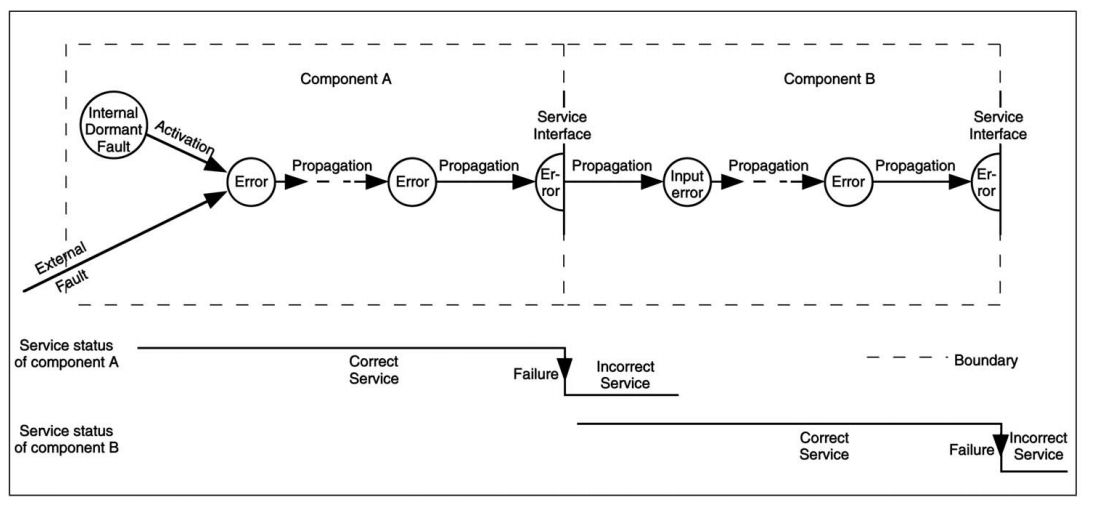


Fig10

* 故障（Fault）产生一个错误后处于被激活（Active）状态，否则是处于休眠（Dormant）状态的。故障激活主要表现为1）一个内部故障一开始失休眠的，但是被内部的程序处理过程或者外部的环境因素激活2）由于外部输入（activation pattern）触发了一个休眠的故障被激活。系统内部的故障就在激活和休眠之间来回变化。
* 错误（Error）在一个部件内部的传播是由于内部计算过程（computation process）造成的：一个错误可能会转变成为另外的错误。错误会从部件A通过内部传播方式传播到接受部件A服务的部件B（当一个错误到达部件A的服务接口的时候）。在这个时候，B接收到的A的业务就是不正确的，也就是说部件A的业务失效就是部件B的一个外部故障（通过A、B之间的业务接口）。
* 业务失效发生在当一个错误传播到系统的业务接口引发了向用户提供不正确的业务的时候。一个部件的失效会在包含着各部件的系统的内部产生一个永久的或者临时性的故障（Fault）。一个系统A的失效会产生会对另一个接受本系统（A）业务服务的系统B产生一个暂时的或者永久的外部故障（external fault）。

上述的机制可以用Fig11的中的威胁链（chain of threats）来表达：

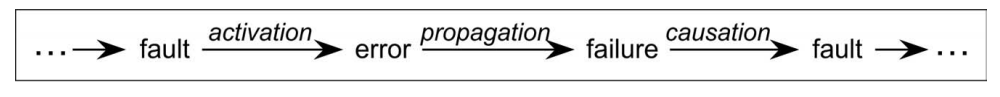


Fig11

上图中的箭头代表故障、错误、失效之间的因果关系（faults, errors, failures）。此处需要说明一下，对于上述过程来说，一个失效可能是由几个错误造成的。值得强调的是，从上述描述的机制来说，传播链的实例化主要是通过部件或者系统之间的交互，部件加入系统，以及一个系统的变更或者创建而产生的。在Fig12给出了一些具体实例，通过这些例子我们可以不难理解到故障休眠性在不同的故障类型以及不同的系统负载下可能会有很大的不同。

一个故障产生一个或者多个错误的激活方法（activation pattern）的测试能力叫做故障激活重复性（fault activation reproducibility）。根据激活重复

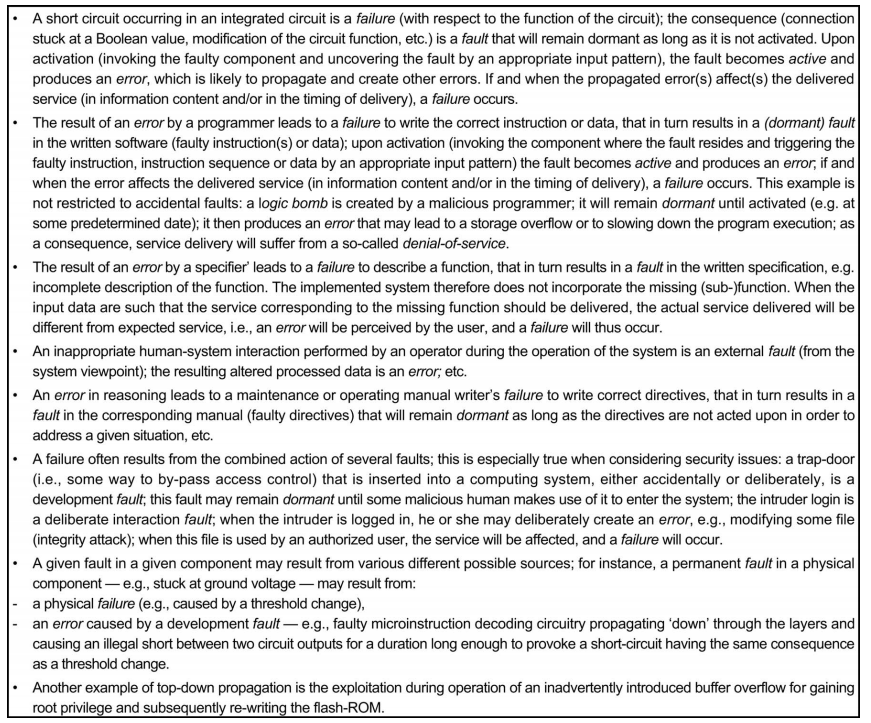


Fig12

性，故障可以分为1）硬故障（Solid、Hard），这种故障的故障激活是可重复的 2）软故障（Soft、Elusive），这种故障的故障激活是无法系统性重复的。大部分在大型复杂软件系统里面残留的开发故障（residual development faults）都是很难再现的（elusive）故障：这也就充分表明，这些故障的激活条件依赖于外部请求和内部状态的非常复杂的组合关系，发生概率低，很难再现。其他的一些例子如下：

* 半导体存储中的模式敏感性故障（pattern sensitive），硬件部件参数的改变（温度变化、寄生电容的时间延迟等）
* 一些特殊的情况，比如系统负载超过了特定的阈值，可能会产生时间的边界效应，同步和时序问题等

基于难以复现的开发故障和瞬态的物理故障的相似性，我们把他们一起归类为**间歇性故障**（intermittent faults）。这类间歇性故障产生的错误成为软错误（Soft Error）。图Fig13总结了上述内容。

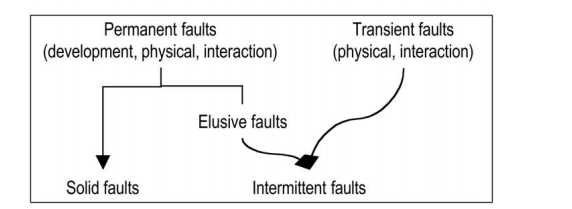


Fig13

多个故障/失效同时发生的情况很常见。系统的失效往往是因为多个并存故障导致的错误的**事后检查**（later examination）。给定一个故障边界的系统，一个单一的故障**（single fault）**是由一个不利的物理事件或者有害的人的行为造成的。群体故障**（Multiple faults）**是两个或者多个并发的，重叠的、或者先后发生的（但是他们导致的Error是重叠发生的）的故障组成的。对于群体故障中的故障之间的关系可以分为1）独立性故障（independent faults），每个故障是由不同的原因引发的2）关联性故障（related faults），多个故障是由同一个原因引发的。关联性故障通常会产生相似的错误（similar errors），这些错误无法通过检测机制进行区分；独立性故障通常会产生不同的错误（distinct errors）。然而也存在独立性故障（尤其是遗漏性故障）产生相似错误的情况，关联性故障产生不同的错误的情况。相似的错误的引发的失效叫做**共模失效（common-mode failures）。**

关于威胁、故障、错误、失效（threats, fault，error，failure）的名词解释附加说明如下：

* 威胁（threats）通常指的是故障（fault）和错误（error）。失效（failure）在安全（Security）领域比起它的通用定义有更加广义的概念，也就是在本质上也保留了潜在的通用解释。在我们的术语定义当中，它既包含了这种潜在方面（例如故障还未激活，业务失效还未影响可信）以及现实方面（故障被激活，业务失效已经发生）。在安全领域。一个恶意的外部故障叫做攻击（attack）。
* 本文中对故障、错误和失效的专有使用并不排除在特殊情况下使用其他简单明了地指明特定威胁类别的词语。尤其适用于故障(例如bug, defect, deficiency, flaw, erratum)和失效(例如breakdown, malfunction, denial-of-service)。
* 由特定术语故障、错误和失效组成的分类，简单地考虑了常见的用法：1）故障预防，容错，诊断（ fault prevention, tolerance, and diagnosis）2) 错误检测与纠正（error detection and correction）3) 失效率（failure rate）

# 4 可信性、安全性的特征

## 4.1可信性、安全性的定义

在2.3节，我们提供了两种不同的可信性（Dependability）的定义：

* 原始定义：能够提供可以让人可信任的业务
* 备选定义：一个系统能够避免频繁和不可接受的业务失效的能力

原始定义是一个通用的定义，目标在于总结了可用性、可靠性、功能安全性、可维护性、完整性（availability, reliability, safety, maintainability、integrity）这几个可信性特征的经典概念。然而备用定义来自于如下的几个争议。一个系统会通常发生失败 那么1）这是系统是否是可信的呢？3）系统什么时候会变得不可信呢？此外可信性失效（dependability failure）的概念从可信性的概念中演化出来的，为的是建立和开发过程失效相关的关系。

可信性在其他的标准中的定义与我们本文有所不同，举两个典型的定义：

* 一个集合用语，用来描述可用性、性能以及他们的影响因素：可用性、性能、可维护性能（maintainability performance）和维修支撑性能（maintenance support performance）
* 在规定的操作与环境条件下，规定的时间（给定的时间、持续的时间），系统能够提供可以依赖的正确任务的程度

第一个定义（ISO）是以可用性（availability）为中心来定义的。这个定义可以追溯到CCITT组织，因为在那个时候可用性是一个电信公司的主要问题。然而授予可用性一个通用的特征的意愿变得重要了，自从它已经超越了最开始定义的可用性的范围，关联到了可靠性（reliability）和可维护性的时候。从这个方面来说，ISO/CCITT组织在附录引文[26]中定义了可信性：系统会按照我们想要的操作的概率。在附录引文[29]中的定义提到了依赖（reliance）的概念，和本文的定义非常接近了。

在安全性（Security）世界的术语就有着更丰富的历史了。很长一段时间的安全研究和实践的社区都使用计算机安全性、通信安全性、信息安全性以及信息保险（assurance）这个概念表达安全性。大部分情况下都没有直接参照可信性的定义。尽管如此，所有上述的表达方式都可以理解为三个主要的安全特征：机密性、完整性、可用性（confidentiality, integrity, and availability）。

安全性（Security）没有被归为可信性（dependability）的一个特征。主要是安全通用定义的视角把安全定义为一个组合（composite）概念，也就是说定义为机密性（confidentiality），阻止未授权的信息暴露，一致性（integrity），阻止未授权的信息修改或者删除，阻止未授权的信息持有，可用性（availability）的组合。安全性在本文中统一的定义为系统状态不存在未授权的接入或者处理。可信性和安全性的关系如同Fig14所示。

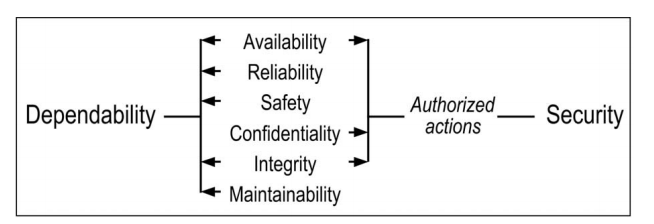


Fig14

## 4.2 依赖和信任（Dependence and Trust）

在2.3节中我们已经定义过依赖和信任的概念，就是A系统对B系统的依赖（dependence）代表着A系统的可信性被（dependability）B系统影响的程度。信任（Trust）是指可接受的依赖性。

一个系统A对另一个系统B的依赖程度的范围可以从完全依赖（total dependence，B的任何失效都会导致A失效），到完全无关（complete independence，B的任何失效都不会导致A失效）。如果有理由相信系统B的可信性不足以支撑A的可信性，那么B的可信性需要加强，A对B的依赖程度降低或者需要额外的故障容错机制（fault tolerance）。我们定义的依赖性和附录引文[50], [14]定义的依赖性关系（relation depends upon），其定义为如果部件B的业务的正确性（correctness）是部件a的业务的正确性的必要条件，则部件a依赖于（depends upon）部件b。然而这种关系被狭义表达为正确性（correctness）而不是可信性（dependability），因此是一个二元的（binary），然而我们的依赖性（dependence）的概念是建立在可度量的空间的（measurable space）。

对于可接受的依赖性，我们的意思是依赖性（例如A On B）和判断力（就是说这样的依赖等级是可接受的）有关。对于B系统的这样的一个判断力（A做出的或者代表A的）可能是非常明确地甚至于是建立在A和B的契约的基础上的，也可能是含蓄的甚至于欠考虑的。实际上可能是迫不得已的---对于A来说没有选择只能依赖与B。因此A对B的信任度，不需要假定B失效的责任（例如提供容错措施）。实际上A没有提供容错B失效的措施的程度是A（可能含蓄的或者欠考虑的perhaps unthinking or unwilling）对B的信任度的一个度量。

## 4.3 可信性和安全性的特征

在2.3节中已经定义了可信性和安全性的特征会取决于一个给定系统的应用（application intended）而具有不同的重要性：可用性，完整性，可维护性（Availability, integrity, and maintainability）通常是必须的，尽管不同的应用程度有所不同；然而可靠性、功能安全性、机密性（reliability, safety, and confidentiality）对于某些应用来说却不是必须的。一个系统拥有可信性和安全性特征的程度应该是相对的、概率性的、可检测的而不是绝对的、固定的：这主要是基于故障发生的不可避免性，系统不可能永远可用、可靠性、功能安全（safe）、系统安全（secure）。

完整性（integrity）的定义是不当的系统状态改变的缺失---超过了通常的定义，那就是1）仅仅与授权操作的概念有关 2）聚焦在信息上（阻止未授权的信息修改或者删除[12]，确保被认可的的信息变更[33]）:1）自然来说，当一个系统实施授权操作策略，不当（“improper”）包含未授权（“unauthorized,”）2）不当的变更（“improper alterations”）包括为了阻止（纠正）信息升级的动作 3）系统状态包含系统的修改或者损坏。

可维护性的定义故意的超越了纠正性和预防性维护的概念，包含了在第3章定义的其他形式的维护动作，例如自适应和增强性的维护。自主计算[22]的概念（autonomic computing）的主要目标是对于大型网络计算系统提供较高的可维护性，尽管本身系统的管理是自动化的。

除了第2章定义的基础特征以及上面的讨论外，其他的辅助性的特征可以定义用来优化或者专业化基础特征（primary）。辅助特征的一个典型例子就是**健壮性（robustness）**，针对外部故障的可信性，体现一个系统对特定的故障的**抵抗**（reaction）能力。

安全性当中有很多的辅助特征的定义，主要是基于不同的信息类型进行区分的。

* 可审计性（accountability）：人员操作记录识别的完整性和可用性
* 真实性（authenticity）：消息内容和来源的完整性
* 不可复制性（nonrepudiability）：信息发送者或者信息接收者识别的完整性和可用性

安全策略（security policy）的概念指的是根植在一个系统[47]或者组织里面以安全为动机的约束限制的集合。这种约束限制主要是通过技术的、管理的或者操作控制的方式来实施的，策略中也有可能规定这些控制是如何实现的。实际上安全策略也是系统规格的一部分，如果缺失的话，会被认为是安全性失效（security failure）。在实践当中安全策略是多层级的相对于系统的多层级而言。

对于任何的可信性和安全性规格集合来说，完整性，一致性，准确性是非常重要的（completeness, consistency, accuracy）。

可信性和安全性的种类（Dependability and security classes）通过失效频率、严重性、中断持续时间、应用所涉及到得特征的分析来进行定义的。这个分析可以直接或者间接的使用风险分析（risk assessment）方法[25] [58] [32]。

不同的可信性和安全性特征上面重点的不同将直接影响建立可信和安全系统的技术的（fault prevention, tolerance, removal, and forecasting：故障预防、容错、清楚、预测）取舍。这些问题当中最困难的是不同的特征之间是有矛盾的和冲突的（例如可用性和功能安全，可用性和机密性等等），需要找到一个折中点（trade offs）。

## 4.4 可信性，高自信、可生存性、可信赖性

其余的和可信性相类似的概念例如high confidence, survivability, and trustworthiness。他们与可信性之间的比较如图Fig15所示，通过逐项对比可以得出一个结论，这四个概念在目标的本质上是相等的并且有着同样的威胁。

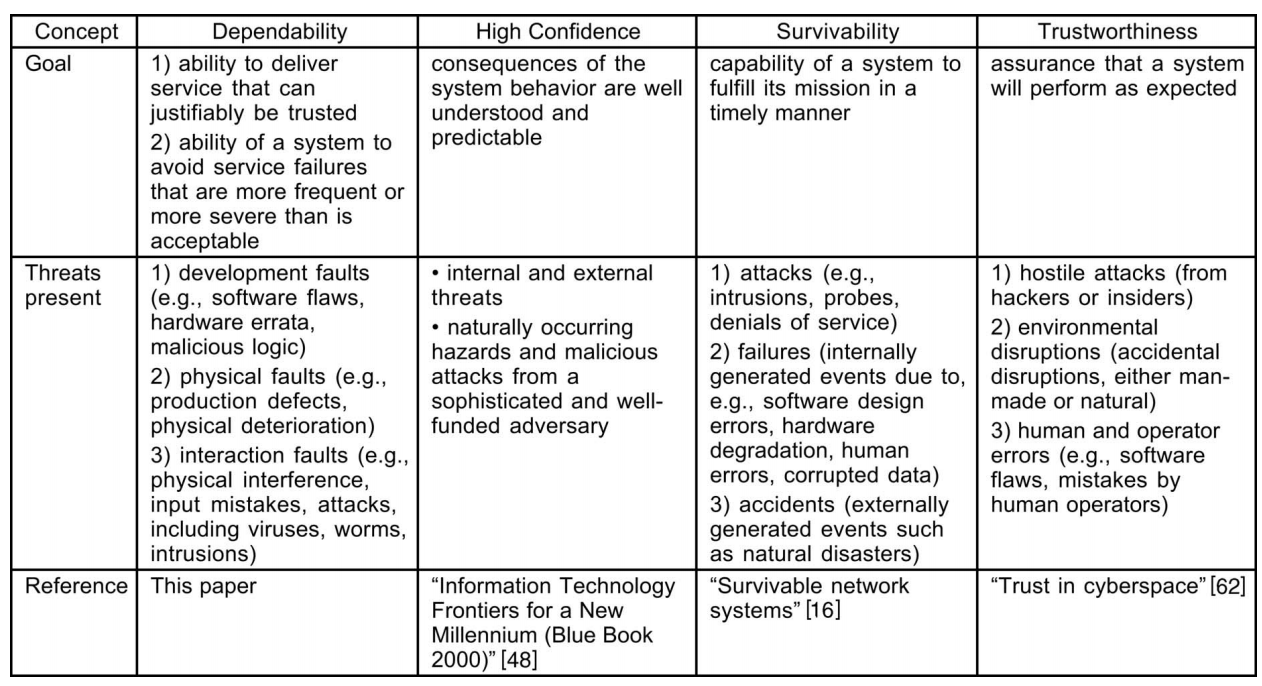


Fig15

# 5 建立可信性、安全性的方法

在本章我们将依次介绍故障预防，故障容错，故障清除，故障预测这四个方法，本章结束于这四个方法的关系描述。

## 5.1 故障预防（Fault Prevention）

故障预防是通用的工程能力的一部分，在本文中不会重点介绍，但是故障预防的一些方面对于可信性和安全性是直接有趣的，可以根据3.2节的故障分类进行讨论。

预防开发故障是一个开发方法论的明显的目标，对于软件（例如信息隐藏，模块化，使用增强类型的编程预研等）和硬件（例如设计规则）都是如此。减少在开发过程中引入系统的故障的数量的开发过程改进是向前一步，这一步主要是基于产品缺陷的统计，通过过程修改（process modifications）[13][51]来删除故障发生的根因。

## 5.2 故障容错（Fault Tolerance）

### 5.2.1 故障容错技术

对于故障容错（Fault Tolerance）[3]，其主要目的是避免失效的发生，主要是通过错误检测（error detection）和系统恢复（system recovery）来实现的。图Fig16展示了故障容错的各种技术：

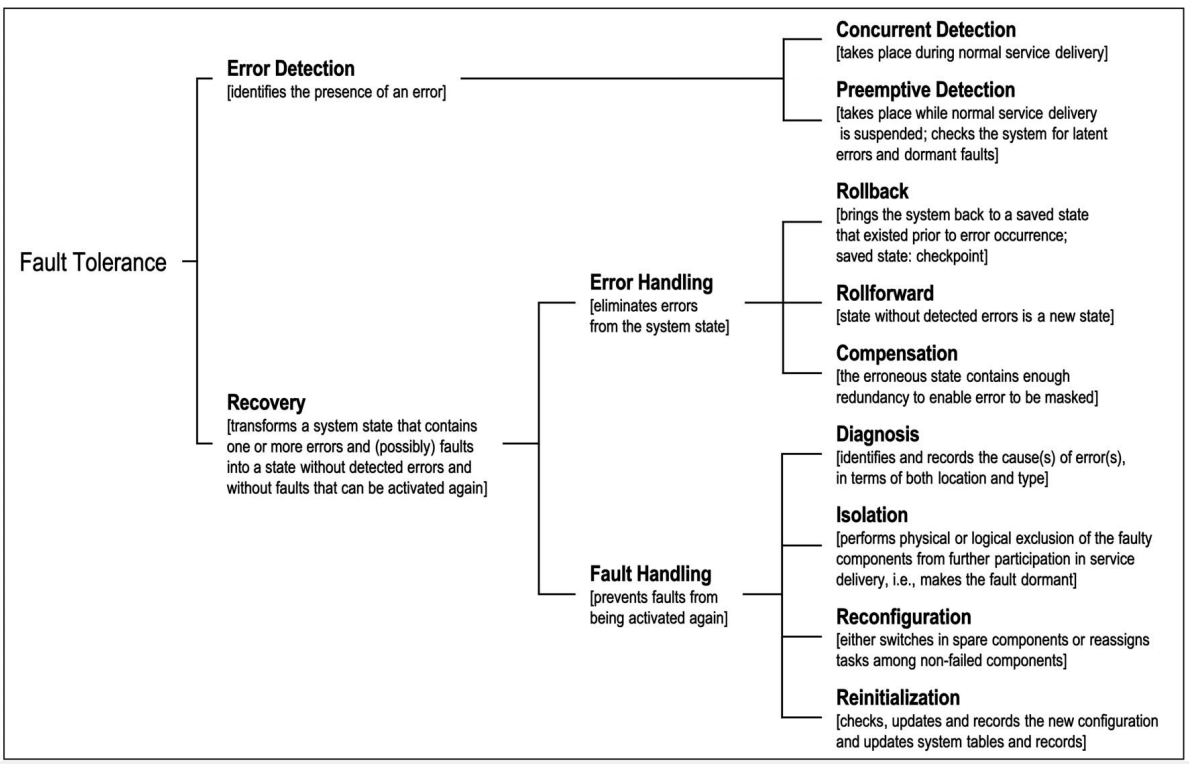


Fig16

通常故障处理（Fault Handling）是在纠正性维护之后，目的在于清楚被隔离的故障。换句话说，区分故障容错和可维护性的主要因素是是否需要外部代理（Agent）的介入。封闭式系统指的是无法进行故障清除（fault removal）的系统。

回滚和前滚（Rollback and rollforward）在错误检测（error detection）后按需调用，然而补偿（compensation）可以按需或者系统性的在事先预定的时间或者事件发生，独立于错误（detected）的存在或者不存在。在故障处理（Fault Handling）的之后的按需错误处理一起构成了系统恢复（system recovery）。因此故障容错的相应策略可以表示为：错误检测和系统恢复（error detection and system recovery），简称为检测与恢复（detection and recovery）。

故障屏蔽（Fault masking）或者简称为屏蔽（masking），来自于补偿的系统性的使用。这样的屏蔽机制将会隐藏可能的累积以及保护性冗余机制的致命缺失。所以故障屏蔽的实践措施通常包含错误检测（也可能是故障处理Fault Handling），接下来是屏蔽和恢复（masking and recovery）。值得注意以下几点：

* 回滚和前滚（Rollback and rollforward）并不是相互独立的，回滚通常优先尝试，如果错误持续的话，尝试前滚的方式。
* 间歇性（Intermittent）的故障不需要隔离（isolation）或者重新配置（reconfiguration）。识别一个故障是否是间歇性的方式要么通过错误检测（错误是否再现）要么通过前滚方式的故障诊断（fault diagnosis）
* 错误检测后可以直接进行故障处理（Fault Handling），不需要先尝试错误处理（error handling）。

|  |
| --- |
|  |

主动性的（Preemptive）的错误检测和处理一般在故障处理之后，通常是在系统处于上电状态。也可能在操作期间进行，体现为不同的形式例如空间检查、内存扫描，程序审计（spare checking, memory scrubbing, audit programs）或者叫做软件恢复（software rejuvenation），目标在于在触发业务失效前清除软件老化（software aging）的影响。Fig17给出了4种典型的和概要的实施故障容错的不同的策略的例子。

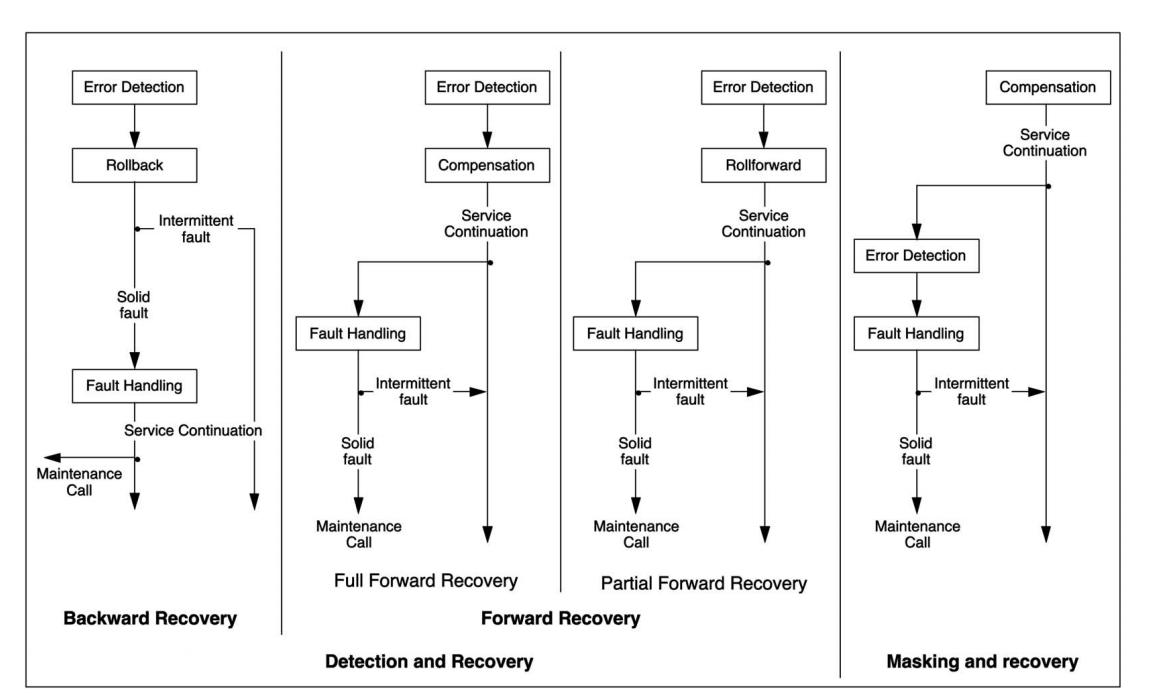


Fig17

### 5.2.2 故障容错实施

错误检测，错误处理，故障处理技术和实施直接关联和强依赖于如下潜在的故障假设：可以真正进行容错的故障种类依赖于这样的故障假设，在开发过程中考虑的，并依赖于故障产生和激活过程的冗余机制的（independence）独立性。广泛使用的获得故障容错的的方法是通过多种途径串行或者并行的实施大量的计算。当预见可以实施一个物理故障的容错时，其途径可以是基于硬件部件独立故障的基础上的假设的同一性设计（identical design）。这样的方法（回滚）[23][28]被证明对难以发现的开发故障比较有效，却不适合于硬开发故障（solid development faults），这种故障每一种都需要通过独立的设计来实现[57][4]，也叫作**设计多样性**（design diversity）[6]**。**

部件内必须要的功能处理能力连同并发的错误检测机制的提供导致了自检部件（self-checking component）概念的出现。无论在软件或者在硬件领域，自检部件的一个重要的好处是能够给出错误限制区[63]的清晰的定义（error confinement areas）。

非常显而易见的不是所有的故障容错技术都是有效的，我们可以使用覆盖度（coverage）来度量某个故障容错技术的有效性。故障容错的缺点是构成一个苛刻对可信性增长的限制。这些缺点主要是以下原因引起的（参见Fig18）：

* 对于影响关于在开发期间声明的故障件假设的故障容错机制的开发故障，就是缺少错误和故障处理覆盖度的结果（error and fault handling coverage）
* 不同于故障真的发生在操作期间产生的故障假设就会导致故障假设覆盖度的不足（fault assumption coverage），会导致要么1）失效的部件没有被考虑到，叫做失效模式覆盖度（failure mode coverage）不足；要么是共模失效的出现当仅考虑独立的失效的时候，叫做失效独立性覆盖度不足（failure independence coverage）。

错误和故障处理覆盖度（error and fault handling coverage）的不足已表明是对可信性提升[8][1]的极端的限制。相似的影响还可能来自于失效模式覆盖度的不足：保守的故障假设（例如拜占庭故障）将会产生较高的失效模式覆盖度，会导致系统可信性和安全性的降低[54]。

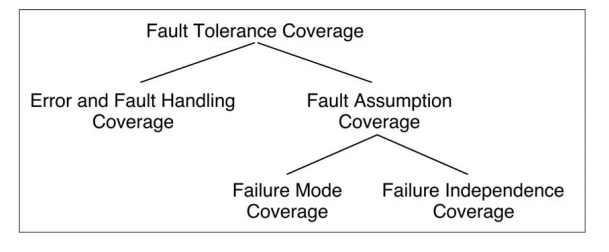


Fig18

另一个重要的问题是协调多个部件的活动来避免错误传播到非失效部件。这个问题在一个部件需要同另一个部件进行信息通信的时候显得尤为重要。单一源（single-source information）信息的典型的例子就是本地传感器数据，本地时钟值，其他部件状态的本地视角等。

故障容错也是一个迭代的概念：非常重要的就是故障容错机制本身也需要被保护以避免容错机制失效。这些保护机制的典型的例子如voter replication, selfchecking checkers, “stable” memory for recovery programs and data。

系统性的故障容错需要增加特定的支撑系统，例如软件狗，业务处理器，指定的通信连接等等（software monitors, service processors, dedicated communication links）

反射（Reflection），是一个明显的和适当的加大一个对象或者软件部件的所有关联动作。例如确保这些动作可以在必要的情况下撤回，可以使用在面向对象的软件以及通过中间件提供[17]。

故障容错适用于所有类型的故障。对于入侵的保护在传统上涉及到密码学和防火墙。一些错误检测机制通常直接针对恶意故障和非恶意故障。入侵检测通常是通过likelihood checks[18][15]。建议的容错的方法和方案如下：

* 入侵和物理故障通过信息分解和分散fragmentation and dispersal [20][56]
* 恶意的逻辑和病毒通过控制流检查control flow checking[35]或者设计差异化（diversity）[36]
* 入侵，恶意逻辑，漏洞这类物理的或者开发故障通过服务器差异化（diversity）[68]

最后，值得注意的是1）故障容错有多个相似的概念，例如自修复，自愈，弹性（self-repair, self-healing, resilience）2）基于恢复的计算[19] （recovery-oriented computing）这个词近来被作为一个故障容错方法用来构建整个系统的可信性，比如在由多个计算机系统组成的系统中，单个系统的失效作为故障进行容错。

## 5.3 故障清除（Fault Removal）

在本章我们会讲述在开发过程以及客户使用过程中的故障清除方法。

### 5.3.1 开发过程的故障清除

开发过程的故障清除包括三个步骤：验证、诊断、修正（verification, diagnosis, and correction）。我们聚焦在继续下去的验证，那就是针对系统是否达到了指定属性的检查过程，叫做验证条件（verification conditions）。如果不是的话，将会采取另外两个步骤：1）诊断导致验证条件无法实施的故障，然后采取必要的修正措施。在修正之后，验证过程再次启动来检查故障清除没有导致不希望的后果，在这个阶段实施的验证叫做非回归性验证（nonregression verification）

我们称检查是否满足规格的过程为验收[7]（validation）。未覆盖的规格类故障可能发生在开发过程的任何阶段，或者是发生在规格定义阶段或者后续的阶段（有证据表明系统没有实现既定的功能或者或者功能实现未使用合理的方式）。

验证技术按照是否运行这个系统进行分类。没有实际执行动作的验证过程称之为**静态验证（static verification）**。总结为如下的两种：

* 对系统本身采用的方式为1）静态分析（static analysis），例如审视、排练，数据流分析，复杂度分析，摘要解析，编译器检查，漏洞搜索（inspections or walk-through, data flow analysis,complexity analysis, abstract interpretation, compiler checks, vulnerability search）2）原理证明（theorem proving）
* 系统行为模型，这个模型通常是一个状态转移模型，也叫作模型检查（model checking）

对系统有实际执行动作的验证过程叫做动态验证（dynamic verification）。对于一个系统的输入可能是抽象的（symbolic，叫做抽象执行symbolic execution）或者实际的验证测试（verification testing）,通常简称为testing。如图Fig19总结了多个验证方法：

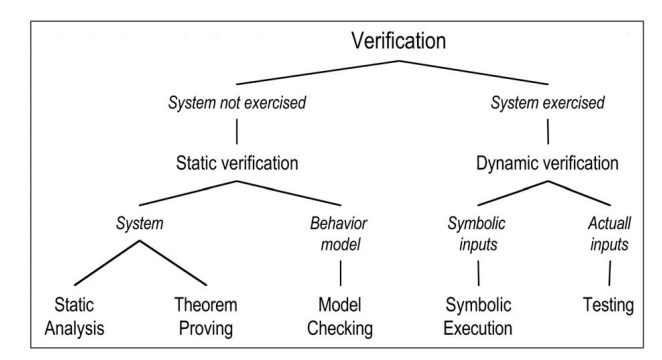


Fig19

针对一个系统的所有输入进行测试的穷尽测试是不现实的，测试的方法的分类依据两个视角：选择测试输入的标准以及测试输入的产生。Fig20总结不同的测试方法:

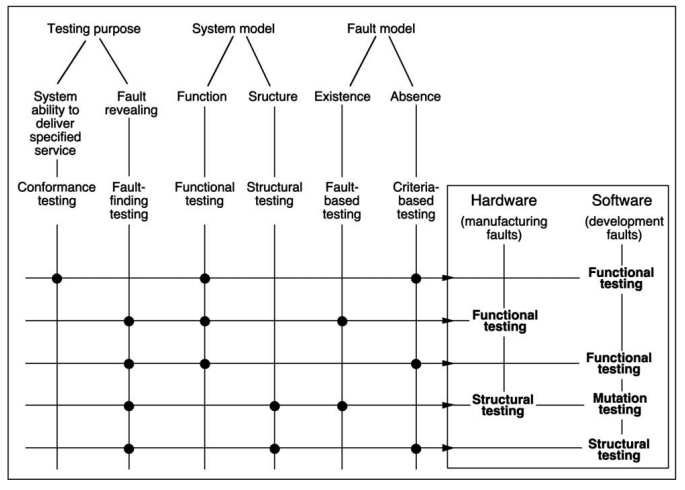


Fig20

这个图的上半部分给出基础的测试方法，下半部分给出了各个基础测试方法的组合。这里的区别主要是软件测试和硬件测试，硬件测试主要的目的在于清除生产故障，软件测试主要目的在于清除开发故障，所以硬件测试通常是基于故障的（fault-based），软件测试通常是基于标准的（criteria-based），但是也有例外的基于故障的混合性测试。

测试输入可能是确定性的也可能是概率性的。

* 确定性测试（deterministic testing）：测试模式是通过认真挑选的方式预先确定的
* 随意性测试或者基于统计的测试（random, or statistical, testing）：测试方案是根据输入域的概率分布定义的。输入数据的数量和分布是由给定的故障模型和标准决定的。

观察测试输出并确定输出是否满足验证条件（verification conditions），就是业界著名的是预言家问题（oracle problem）。验证条件可能适用于整个输出集合也可能是后来的小型功能（例如测试硬件故障时的系统签名，测试软件开发故障时的“partial oracle”）。当测试物理故障的时候，对于一个指定的通过模拟或者参考系统而确定的给定输入序列的被测系统，结果是紧密的（compact）或者非期待的。对于开发故障主要参照规格，它可能是一个原型或者同样的规格的通过设计多样性后的不同的实现(back-to-back testing)。

验证方法可以组合使用的。举个例子，抽象操作（symbolic execution）用来促进测试模式（testing patterns）的确定。理论证明（theorem proving）用来检查无限状态模型[60]（infinite state models）的属性。混合型测试可以用来比较不提供的测试策略[66]（testing strategies）。

验证是贯穿于整个系统开发过程的，因此上面的测试技术适用于各种形式的系统的开发过程的（例如prototype, component）。

上面的测试技术也适用于故障容错机制的验证，尤其是1）正式的静态验证[59]（formal static verification）2）测试在测试模式中的必要的故障或者错误，通常叫做故障注入[2]（fault injection）

验证一个系统的功能没有（cannot do more than）超过预定的规格是非常重要的，因为这关乎到功能安全性（safety）和系统安全性（Security，渗透测试penetration testing）。

设计一个易于验证的系统叫做可验证性设计（design for verifiability），这种方法在针对硬件故障的硬件领域做的比较好。有一个相似的技术叫做可测试性设计（design for testability）。

### 5.3.2 使用过程的故障清除

在使用过程中的故障清除指的是修正或者主动性维护（corrective or preventive maintenance）。修正性维护的目标在于清除已经产生被报告一个或者多个错误（Errors）的故障（Faults）。主动性维护的目标在于暴露和清除在正常操作当中存在会引发错误的故障，这种故障包含1）上一次主动性维护操作之后的物理故障2）已经在其他类似系统中引发错误的开发故障。对于开发故障的纠正性维护过程为：这个故障在故障清除动作完整之前首先要被隔离（替代方法，补丁等）。这种形式的维护同时适用于非故障容错的系统和故障容错的系统，这些熊可以进行在线维护（without interrupting service delivery）和离线维护（during service outage）。

## 5.4 故障预测（Fault Forecasting）

故障预测是基于故障发生概率或者活动性的系统行为估计（evaluation of the system behavior）。估计包含如下两个方面：

* 定性估计（qualitative, or ordinal, evaluation）：主要目标在于对会导致系统失效的失效模式以及失效模式组合的识别、分类、定级。
* 定量估计（quantitative, or probabilistic, evaluation）：主要目标在于评估当某些特征满足情况下的系统失效概率，这些特征被视为度量（measures）

定性或者定量评估方法可能是独特的（例如FMEA主要适用于定性分析，马尔科夫链和随机离散网络主要适用于定量分析）也可能是都适用于这两种评估方式（可靠性框图RBD，故障树FTA）。

目标在于获取概率估计值（probabilistic estimates）的两种概率性故障预测方法是建模（modeling）和估计测试（evaluation testing）。这两个方法是互补的，因为建模需要建立在模型化过程（failure process, maintenance process, system activation process）的数据的基础上，这些数据要么是通过测试获取的要么是通过失效数据处理得到的。

建模可以从以下几种故障类型来实现：物理故障、开发故障，物理故障和开发故障的组合故障。尽管建模通常是针对非恶意的故障，但是针对恶意故障的建模也是值得去研究的[49][61]。建模分为如下两个阶段：

* 基于基础随机过程的系统模型的建立（construction），这个随机过程是基于系统部件行为和部件交互行为。这个基础随机过程和失效以及业务恢复措施有关，恢复措施包含修复以及可能是系统服务生命周期、活动阶段
* 模型的处理过程（Processing）主要用于获得系统可信性度量的值或者表达式

通常几个业务可以被区分为两个或者多个业务模型，范围从系统的全部能力到应急模式。这些模型区分会导致完整业务输出越来越少。可信性的性能相关的度量归结到一个完成度概念[45][64]（performability）。

可靠性增长模型（Reliability growth models），无论对于软件或者硬件或者软硬件一体来说，通常都是用来基于过去的失效数据对未来的可靠性进行预测。

估计测试(Evaluation testing)采用5.3.1节中的视角可以描述为适应性、功能性、非基于故障的、静态的测试（conformance, functional, non-fault-based, statistical, testing），尽管它的主要目标不是验证（verifying）一个系统。一个主要的问题是它的输入剖面是它的操作剖面的典型代表。因此估计测试也通常叫做操作性测试（operational testing）。

当评估一个容错系统，错误和故障处理机制的覆盖度会强烈影响可信性的度量[8[1]。这个覆盖度的评估可以通过建模或者通过测试获取。

可行性和安全性基准（benchmark）的概念，指的是在出现故障的时候对系统的行为进行评估度量的过程，这也使得各种故障预测技术结合在一起形成整体框架。这样的基准使得1）对系统可信性和安全性的刻画（characterization）2）一个或者多个特征的可替代的或者竞争性解决方案的比较[37]（comparison）。

## 5.5 故障预测、清除、容错、预防之间的关系

在第二章中给出的故障预防，清除，容错，预测定义中的各种实施方法，很难完全达到实际的目标，因为所有的设计和分析活动都是人进行的而无法达到完美。这些不完美正好解释了为什么只有将上述活动的组合，最好在设计和实现过程的每一步中使用，才能最好地打造一个可靠安全的计算系统。这种关联关系可以被描述为如下内容：尽管通过开发原则和架构规范实现故障预防（Prevention），但是故障还是会发生，因此就需要故障清除（Removal），故障清除自身也是不完美的（例如不会发现所有的故障，清除故障的过程中也会产生故障），使用现成的组件，无论是硬件还是软件都会包含故障，因此故障预测（除了对操作故障的可能后果的分析）就显得非常重要。计算系统的可信性不断提升带来了故障容错的需求，这个也是基于架构规范（construction rules）的基础上的。因此再一次需要应用故障清除和故障预测到故障容错自身的机制上。值得注意的是这个过程会比上面出的更加递归的：当前的计算系统是非常复杂的，它的设计和实施需要软件和硬件工具来使得开发过程更加高效（广泛意义来说，包含在可接受时间范围内成功的能力）。这些工具本身也需要时可信的和安全的。

前面的原因已经展示了故障预测和故障清楚地闭环交互。促使他们聚集到可信和安全分析（dependability and security analysis），目的在于达到自信（reaching confidence）在输出可信任的业务的能力情况下。然而故障预防和故障容错在一起组成**可信和安全规定（**dependability and security provision），目的在于提供可信任的业务的能力。这些方法的分组1）故障预防和故障清除归类为故障避免（fault avoidance），目标是如何建立无故障的系统2）故障预测和故障容错归类为故障接受（fault acceptance），目标在于在故障情况下系统如何生存下来。Fig21展示了上述方法的分类。值得注意的是当聚焦在安全领域，这样的分析通常叫做安全评估（security evaluation）。

除了突出评估故障容错的过程和机制重要之外，故障清除和故障预测也要考虑作为可信性分析的两个组成成分，会促进对覆盖度（coverage）的概念的理解，也因此引入了一个重要的递归问题：对评估的评估（the assessment of the assessment），或者使用什么样的工具和方法来达到自信，这些工具和方法使用在建立自信的系统里。覆盖度（coverage）在这里指的是一种代表性的度量，它的定义是非常通用的，但是它可能会变得精确通过定义应用的范围的情况下，例如软件测试的覆盖度是关于软件上下文、控制路径的覆盖等，集成电路测试的覆盖度是关于故障模型的覆盖，故障容错的覆盖度是关于故障类型的覆盖。

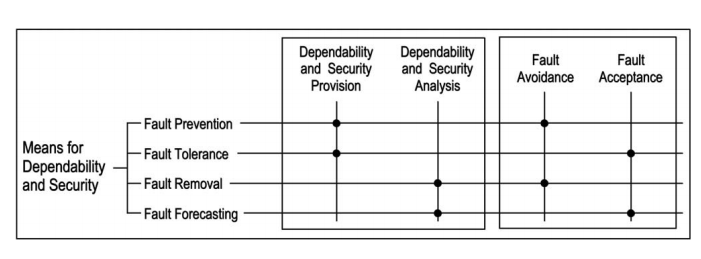


Fig21

评估一个系统是否真的可信和安全（所提供的业务是真正可以被信任的）超出了在前面的章节提到的分析技术，至少基于如下三个原因和限制：

* 设计覆盖度的精确检查或者关于现实的验收假设（在故障容错机制设计过程中的故障猜测）都会包含一门知识或者掌握一项技术，对系统的预期利用等会超过截止目前我们可以通常获取的。
* 评估一个系统可信性的某些特征并且尤其是关于特定类型故障下的安全在当前被认为是难以实现的或者产生无意义的结果因为概率理论基础（probability-theoretic bases）是不存在的或者没有被广泛接受的。典型的例子是偶然性开发故障下的功能安全（safety），故意故障下的系统安全（security）。
* 和任何的系统一样，关于规格的分析都是有可能包含故障的

在大量的针对这些事件这些状态的结果中，让我注意到：

* 当我们评估一个系统的时候，重点放在开发过程上，在开发过程中采用的方法和技术。一个评级（grade）被指定并且传递给这个系统，根据1）开发过程所使用的方法和技术的本质2）效用的评估
* 容错系统的规格中给出的需要容错的故障类型和数量的列表。这样的规格不是必要的如果上面提到的限制能够解决。

# 6 结论

越来越多的个人和组织都在开发或者获取复杂的计算系统，这个系统的业务需要更好的信任度---无论这个系统是否是服务于取款设备、卫星发射装备，飞机、核反应堆，放射性治疗设备或者为了保护敏感数据的机密性。在不同的条件下，业务属性的聚焦点将会是不同的，例如能够取得的平均实时反应时间，产生所需要的结果的可能性，避免失效（对系统是灾难性的）的能力，针对蓄意入侵的阻止程度。同时考虑可信性和安全性提供了一个很容易达到的方法，这个方法是在一个统一的概念框架内纳入各种各样的考虑。它包含类似于可用性、可靠性、功能安全性、机密性、完整性、可维护性这样的特定的属性（availability, reliability, safety, confidentiality, integrity, maintainability），它也提供了一些根据客户的需要对这些属性进行合适平衡的（appropriate balance）能力。

在本文中给出的这些概念的长处是它的综合的特质。这将使得可用性、可靠性、功能安全性、机密性、完整性、可维护性这些经典的概念被注入了客观性。故障-错误-失效模型的核心是理解和掌握影响系统的各种威胁（threats）

并且形成了针对这些威胁的统一表达，当需要在定义的各种故障下仍然维持他们的规格。这个模型定义了取得可信性和安全性的方法是非常有效的，因为这些方法是根据可信性的特征而彼此正交的相对于以前的分类而言。开发任何的真实系统都需要在这些特征中进行折中，因为这些属性之间可能是冲突的。针对第二章中的可信性和安全性树的改进参见Fig22：

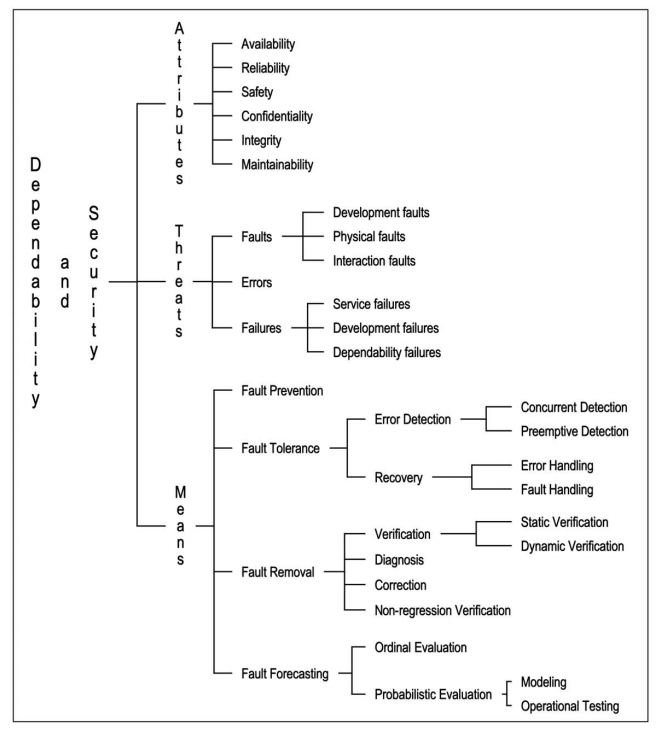


Fig22

附录：

[1] T.F. Arnold, “The Concept of Coverage and Its Effect on the Reliability Model of Repairable Systems,” IEEE Trans. Computers, vol. 22, no. 6, pp. 251-254, June 1973.

[2] D. Avresky, J. Arlat, J.C. Laprie, and Y. Crouzet, “Fault Injection for Formal Testing of Fault Tolerance,” IEEE Trans. Reliability, vol. 45, no. 3, pp. 443-455, Sept. 1996.

[3] A. Avi zienis, “Design of Fault-Tolerant Computers,” Proc. 1967 Fall Joint Computer Conf., AFIPS Conf. Proc., vol. 31, pp. 733-743, 1967.

[4] A. Avi zienis and L. Chen, “On the Implementation of N-Version Programming for Software Fault Tolerance During Execution,” Proc. IEEE COMPSAC 77 Conf., pp. 149-155, Nov. 1977

[5] A. Avi zienis and Y. He, “Microprocessor Entomology: A Taxonomy of Design Faults in COTS Microprocessors,” Dependable Computing for Critical Applications 7, C.B. Weinstock and J. Rushby, eds., pp. 3-23, 1999.

[6] A. Avi zienis and J.P.J. Kelly, “Fault Tolerance by Design Diversity: Concepts and Experiments,” Computer, vol. 17, no. 8, pp. 67-80, Aug. 1984.

[7] B.W. Boehm, “Guidelines for Verifying and Validating Software Requirements and Design Specifications,” Proc. European Conf. Applied Information Technology (IFIP ’79), pp. 711-719, Sept. 1979.

[8] W.G. Bouricius, W.C. Carter, and P.R. Schneider, “Reliability Modeling Techniques for Self-Repairing Computer Systems,” Proc. 24th Nat’l Conf. ACM, pp. 295-309, 1969.

[9] C. Cachin, J. Camenisch, M. Dacier, Y. Deswarte, J. Dobson, D. Horne, K. Kursawe, J.C. Laprie, J.C. Lebraud, D. Long, T. McCutcheon, J. Muller, F. Petzold, B. Pfitzmann, D. Powell, B. Randell, M. Schunter, V. Shoup, P. Verissimo, G. Trouessin, R.J.

Stroud, M. Waidner, and I. Welch, “Malicious- and AccidentalFault Tolerance in Internet Applications: Reference Model and Use Cases,” LAAS report no. 00280, MAFTIA, Project IST-1999-11583, p. 113, Aug. 2000.

[10] V. Castelli, R.E. Harper, P. Heidelberger, S.W. Hunter, K.S. Trivedi, K. Vaidyanathan, and W.P. Zeggert, “Proactive Management of Software Aging,” IBM J. Research and Development, vol. 45, no. 2, pp. 311-332, Mar. 2001.

[11] “Termes et De´finitions Concernant la Qualite´ de Service, la Disponibilite´ et la fiabilite´,”Recommandation G 106, CCITT, 1984.

[12] Information Technology Security Evaluation Criteria, Harmonized criteria of France, Germany, the Netherlands, the United Kingdom, Commission of the European Communities, 1991.

[13] R. Chillarege, I.S. Bhandari, J.K. Chaar, J. Halliday, D.S. Moebus, B.K. Ray, and M.-Y. Wong, “Orthogonal Defect Classification-A Concept for In-Process Measurements,” IEEE Trans. Software Eng., vol. 18, no. 11, pp. 943-956, Nov. 1992.

[14] F. Cristian, “Understanding Fault-Tolerant Distributed Systems,” Comm. ACM, vol. 34, no. 2, pp. 56-78, 1991

[15] H. Debar, M. Dacier, M. Nassehi, and A. Wespi, “Fixed vs. VariableLength Patterns for Detecting Suspicious Process Behavior,” Proc. Fifth European Symp. Research in Computer Security, Sept. 1998.

[16] R.J. Ellison, D.A. Fischer, R.C. Linger, H.F. Lipson, T. Longstaff, and N.R. Mead, “Survivable Network Systems: An Emerging Discipline,” Technical Report CMU/SEI-97-TR-013, Carnegie Mellon Univ., May 1999.

[17] J.C. Fabre, V. Nicomette, T. Perennou, R.J. Stroud, and Z. Wu, “Implementing Fault Tolerant Applications Using Reflective Object-Oriented Programming,” Proc 25th IEEE Int’l Symp. FaultTolerant Computing (FTCS-25), pp. 489-498, 1995.

[18] S. Forrest, S.A. Hofmeyr, A. Somayaji, and T.A. Longstaff, “A Sense of Self for Unix Processes,” Proc. 1996 IEEE Symp. Security and Privacy, pp. 120-128, May 1996.

[19] A. Fox and D. Patterson, “Self-Repairing Computers,” Scientific Am., vol. 288, no. 6, pp. 54-61, 2003.

[20] J.M. Fray, Y. Deswarte, and D. Powell, “Intrusion Tolerance Using Fine-Grain Fragmentation-Scattering,” Proc. 1986 IEEE Symp. Security and Privacy, pp. 194-201, Apr. 1986.

[21] “Fundamental Concepts of Fault Tolerance,” Proc. 12th IEEE Int’l Symp. Fault-Tolerant Computing (FTCS-12), pp. 3-38, June 1982.

[22] A.G. Ganek and T.A. Korbi, “The Dawning of the Autonomic Computing Era,” IBM Systems J., vol. 42, no. 1, pp. 5-18, 2003.

[23] J.N. Gray, “Why do Computers Stop and What Can Be Done About It?” Proc. Fifth Symp. Reliability in Distributed Software and Database Systems, pp. 3-12, Jan. 1986.

[24] J. Gray, “Functionality, Availability, Agility, Manageability, Scalability—the New Priorities of Application Design,” Proc. Int’l Workshop High Performance Trans. Systems, Apr. 2001.

[25] R. Grigonis, “Fault-Resilience for Communications Convergence,” Special Supplement to CMP Media’s Converging Comm. Group, Spring 2001.

[26] J.E. Hosford, “Measures of Dependability,” Operations Research, vol. 8, no. 1, pp. 204-206, 1960.

[27] Y. Huang, C. Kintala, N. Kolettis, and N.D. Fulton, “Software Rejuvenation: Analysis, Module and Applications,” Proc. 25th IEEE Int’l Symp. Fault-Tolerant Computing, pp. 381-390, June 1995.

[28] Y. Huang and C. Kintala, “Software Fault Tolerance in the Application Layer,” Software Fault Tolerance, M. Lyu, ed., pp. 231- 248, 1995.

[29] Industrial-Process Measurement and Control—Evaluation of System Properties for the Purpose of System Assessment, Part 5: Assessment of System Dependability, Draft, Publication 1069-5, Int’l Electrotechnical Commission (IEC) Secretariat, Feb. 1992.

[30] “Functional Safety of Electical/Electronic/Programmable Electronic Safety-Related Systems,” IEC Standard 61505, 1998.

[31] “Quality Concepts and Terminology,” part 1: Generic Terms and Definitions, Document ISO/TC 176/SC 1 N 93, Feb. 1992.

[32] “Common Criteria for Information Technology Security Evaluation,”ISO/IEC Standard 15408, Aug. 1999.

[33] J. Jacob, “The Basic Integrity Theorem,” Proc. Int’l Symp. Security and Privacy, pp. 89-97, 1991.

[34] J. Johnson, “Chaos: The Dollar Drain of IT Project Failures,” Application Development Trends, pp. 41-47, Jan. 1995.

[35] M.K. Joseph and A. Avi zienis, “A Fault Tolerance Approach to Computer Viruses,” Proc. Symp. Security and Privacy, pp. 52-58, Apr. 1988.

[36] M.K. Joseph and A. Avi zienis, “Software Fault Tolerance and Computer Security: A Shared Problem,” Proc. Ann. Joint Conf. Software Quality and Reliability, pp. 428-432, Mar. 1988.

[37] “DBench Dependability Benchmarks,” DBench, Project IST-2000- 25425, K. Kanoun et al., eds., pp. 233, May 2004.

[38] L. Lamport, R. Shostak, and M. Pease, “The Byzantine Generals Problem,” ACM Trans. Programming Languages and Systems, vol. 4, no. 3, pp. 382-401, July 1982.

[39] C.E. Landwher, A.R. Bull, J.P. McDermott, and W.S. Choi, “A Taxonomy of Computer Program Security Flaws,” ACM Computing Survey, vol. 26, no. 3, pp. 211-254, 1994.

[40] J.C. Laprie, “Dependable Computing and Fault Tolerance: Concepts and Terminology,” Proc. 15th IEEE Int’l Symp. Fault-Tolerant Computing (FTCS-15), pp. 2-11, June 1985.

[41] Dependability: Basic Concepts and Terminology, J.C. Laprie, ed.,

Springer-Verlag, 1992.

[42] J.C. Laprie, “Dependability—Its Attributes, Impairments and Means,” Predictably Dependable Computing Systems, B. Randell et al., eds., pp. 3-24, 1995.

[43] N.A. Lynch, Distributed Algorithms. Morgan Kaufmann, 1996.

[44] J. McLean, “A Comment on the ‘Basic Security Theorem’ of Bell and LaPadula,” Information Processing Letters, vol. 20, no. 2, pp. 67- 70, 1985.

[45] J.F. Meyer, “On Evaluating the Performability of Degradable Computing Systems,” Proc. Eighth IEEE Int’l Symp. Fault-Tolerant Computing (FTCS-8), pp. 44-49, June 1978.

[46] J. Musa, “The Operational Profile in Software Reliability Engineering: An Overview,” Proc. Third IEEE Int’l Symp. Software Reliability Eng. (ISSRE ’92), pp. 140-154, 1992.

[47] An Introduction to Computer Security: The NIST Handbook, Special Publication 800-12, Nat’l Inst. of Standards and Technology, 1995.

[48] National Science and Technology Council, “Information Technology Frontiers for a New Millennium,”Supplement to the Prsident’s FY 2000 Budget, 2000.

[49] R. Ortalo, Y. Deswarte, and M. Kaaniche, “Experimenting with Quantitative Evaluation Tools for Monitoring Operational Security,” IEEE Trans. Software Eng., vol. 25, no. 5, pp. 633-650, Sept./ Oct. 1999.

[50] D. Parnas, “On the Criteria to be Used in Decomposing Systems into Modules,” Comm. ACM, vol. 15, no. 12, pp. 1053-1058, Dec. 1972.

[51] M.C. Paulk, B. Curtis, M.B. Chrissis, and C.V. Weber, “Capability Maturity Model for Software,” Technical Reports CMU/SEI-93-

TR-24, ESC-TR-93-177, Software Eng. Inst., Carnegie Mellon Univ., Feb. 1993.

[52] C.P. Pfleeger, “Data Security,” Encyclopedia of Computer Science, A. Ralston et al., eds., Nature Publishing Group, pp. 504-507, 2000.

[53] D. Powell, G. Bonn, D. Seaton, P. Verissimo, and F. Waeselynck, “The Delta-4 Approach to Dependability in Open Distributed Computing Systems,” Proc. 18th IEEE Int’l Symp. Fault-Tolerant Computing (FTCS-18), pp. 246-251, June 1988.

[54] D. Powell, “Failure Mode Assumptions and Assumption Coverage,” Proc. 22nd IEEE Int’l Symp. Fault-Tolerant Computing (FTCS-22), pp. 386-395, June 1992.

[55] “Conceptual Model and Architecture of MAFTIA,”MAFTIA, Project IST-1999-11583, D. Powell and R. Stroud, eds., p. 123, Jan. 2003.

[56] M.O. Rabin, “Efficient Dispersal of Information for Security, Load Balancing and Fault Tolerance,” J. ACM, vol. 36, no. 2, pp. 335-348, Apr. 1989.

[57] B. Randell, “System Structure for Software Fault Tolerance,” IEEE Trans. Software Eng., vol. 1, no. 2, pp. 220-232, June 1975.

[58] “Software Considerations in Airborne Systems and Equipment Certification,”DO-178-B/ED-12-B, Requirements and Technical Concepts for Aviation/European Organization for Civil Aviation Equipement, 1992.

[59] J. Rushby, “Formal Specification and Verification of a FaultMasking and Transient-Recovery Model for Digital Flight Control Systems,” Proc. Second Int’l Symp. Formal Techniques in Real Time and Fault-Tolerant Systems, 1992.

[60] J. Rushby, “Formal Methods and Their Role in the Certification of Critical Systems,” Technical Report CSL-95-1, SRI Int’l, 1995.

[61] W.H. Sanders, M. Cukier, F. Webber, P. Pal, and R. Watro, “Probabilistic Validation of Intrusion Tolerance,” Supplemental Volume Int’l Conf. Dependable Systems and Networks (DSN-2002), pp. 78-79, June 2002.

[62] Trust in Cyberspace. F. Schneider, ed., Nat’l Academy Press, 1999.

[63] D.P. Siewiorek and R.S. Swarz, Reliable Computer Systems, Design and Evaluation. Digital Press, 1992.

[64] R.M. Smith, K.S. Trivedi, and A.V. Ramesh, “Performability Analysis: Measures, an Algorithm, and a Case Study,” IEEE Trans. Computers, vol. 37, no. 4, pp. 406-417, Apr. 1988.

[65] “Dependability Assessment Criteria,” SQUALE project (ACTS95/ AC097), LAAS Report no. 98456, Jan. 1999.

[66] P. Thevenod-Fosse, H. Waeselynck, and Y. Crouzet, “An Experimental Study on Softawre Structural Testing: Deterministic Testing Versus Random Input Generation,” Proc. 21st IEEE Int’l Symp. Fault-Tolerant Computing, pp. 410-417, June 1981.

[67] USA Department of Transportation, Office of Inspector General, “Audit Report: Advance Automation System,”Report AV-1998- 113, Apr. 1998.

[68] A. Valdes, M. Almgren, S. Cheung, Y. Deswarte, B. Dutertre, J. Levy, H. Saı ¨di, V. Stavridou, and T. Uribe, “An Adaptative Intrusion-Tolerant Server Architecture,” Proc. 10th Int’l Workshop Security Protocols, Apr. 2002

[69] E.J. Weyuker, “On Testing Nontestable Programs,” The Computer J., vol. 25, no. 4, pp. 465-470, 1982.  
[70] A. Wood, “NonStop Availability in a Client/Server Environment,” Tandem Technical Report 94.1, Mar. 1994