摘要：许多现代软件系统由同时运行的多个执行流组成，从设计用于利用现代多核架构的功能的应用程序到由部署在不同物理节点上的多个组件组成的分布式系统。我们统称将此类系统称为并发系统。并发系统难以测试，因为源于其并发性质的故障取决于各个执行流程执行的动作的交错。针对这些故障的测试技术必须考虑系统的并发性方面。并行和分布式架构的日益迅速普及导致了并发软件系统的泛滥，以及过去十年中这类系统的测试技术的爆炸式增长。目前缺乏对并发系统的许多测试技术的全面分类，分析和比较限制了对每种方法的优点和缺点的理解，并且妨碍了该领域的未来发展。该调查提供了一个框架，用于捕获可用于测试并发软件系统的技术的关键特征，确定一组分类标准以审查和比较可用技术，并详细讨论它们的优缺点，从而对为未来的发展铺平道路。

Introduction：并发软件系统在几个应用领域中很常见：许多交互式应用程序利用多线程范例将输入输出处理与后端计算分离; 为单个节点开发的应用程序通常利用多个线程在多核架构上实现并行计算; Web应用程序通过客户端和服务器端计算实现客户端 - 服务器通信范例;移动应用程序通常访问数据并与远程服务器交互; 对等服务协调多个计算节点。

并发软件系统由同时执行的多个执行流组成。。。。。。

并发故障本质上是非确定性的，因为它们仅在存在特定交织的情况下发生，并且交织依赖于不受程序直接控制的执行条件。 解决有效探索交织空间问题的测试技术考虑了以下一个或多个活动：（i）生成测试用例，这些测试用例是刺激系统的操作序列;（ii）为执行流选择交错的子集; （iii）使用所选的测试用例和交错执行系统并验证结果。

。。。。。。据我们所知，对并发软件系统领域的进展和结果的精确调查和分类是缺少的。

在本文中，我们提供了对测试并发软件系统的最新技术的全面调查。 我们通过系统地浏览主要出版商和科学搜索引擎来研究最近的文献，并且我们将结果追溯到过去四十年的开创性工作。我们提出了一个总体框架，它捕获了测试并发软件系统问题的不同方面。 我们用它来确定一组分类标准，推动不同方法的调查。 该调查对最先进的技术进行了分类和比较，讨论了它们的优点和局限性，并指出了测试并发软件系统领域的开放性问题和可能的研究方向。

在本文的其余部分安排如下。 第2节介绍了并发系统的背景信息，定义了本文中使用的术语，并提供了一个捕获并发系统测试技术关键方面的通用框架。 第3节介绍了测试并发系统研究的历史趋势，并描述了我们选择相关工作的方法。 第4节介绍了一组标准，用于对并发系统的测试技术进行分类。 第5节对最先进的技术进行了调查和分类。 第6节讨论了文献分析中出现的主要观察结果。 第7节简要概述了本调查范围边界的一些重要领域，第8节总结了论文的贡献。

2.并发软件系统

在本节中，我们定义了分析的范围，并介绍了我们在本文中采用的术语。 为此，我们定义了一个概念框架，用于捕获测试并发软件系统的不同方法的主要元素。 在本文的其余部分，我们使用框架来构建我们的调查。

2.1并发系统

。。。。。

。。。该定义包括在重叠时间帧中执行的流程，例如在多核，多处理器并行和多节点分布式体系结构上执行的并发程序，以及仅在非重叠帧中执行的流程，例如在单个上执行的并发程序 核心架构。 根据特定的体系结构和编程范例，执行流可以具体实现为不同物理机器上的进程，同一机器内的进程或同一进程内的线程，这在现代编程语言（如C ++，Java，C＃和Erlang）中很常见。

我们基于它们采用的机制来区分两类并发系统，以实现执行流，共享内存和消息传递系统之间的交互。 在共享存储器系统中，执行流程通过访问公共存储器进行交互。 在消息传递系统中，执行流通过交换消息进行交互。 消息传递可以由托管在同一物理节点上的执行流程或不同的物理节点（分布式系统）使用。 相反，共享内存机制仅在执行流位于同一节点上时才有可能（如在多线程系统中）。

我们将共享内存建模为一个或多个数据项的存储库。 数据项具有关联的值和类型。 数据项的类型确定允许它采用的值集。 我们使用两个基本操作来模拟执行流f与存储库的交互：写操作wx（v），意味着f将数据项x的值更新为v，读取操作rx(v)，这意味着f读取x的值v。 操作由一个或多个指令组成。 指令是原子的，这意味着它们的执行不能与其他指令交错，而操作通常不是原子的。 该模型捕获简单数据上的操作，如C中的原始变量，以及复杂数据结构（如Java对象）上的操作，其中类型是类，数据项是对象，操作是只能在对象的某些字段上操作的方法。

我们使用两个基本操作对消息传递系统建模：发送操作Sf(m)将消息m发送到执行流f，接收操作Rf(m)，其从执行流f接收消息m。 消息传递可以是同步的也可以是异步的。 向执行流f0发送同步消息sf0(m)的执行流f在继续之前必须等待f0接收消息m，而将异步消息sf0(m)发送到执行流f0的执流f可以立即进行而不用等待f0接收m。

通过将发送原语建模为共享队列上的写操作和接收原语作为同一共享队列上的读操作，可以将消息传递范例映射到共享内存范例。 因此，在不失一般性的情况下，我们在本调查中提供的大多数定义和示例中都提到了共享内存系统。

2.2执行流的交错

并发系统的行为不仅依赖于。。。而且依赖于执行流之间的交错。。。

按照我们在第5节中讨论的绝大多数方法，我们在顺序一致模型的假设下介绍了并发的主要概念[7]。 此模型保证并发系统中的所有执行流遵循相同的指令顺序，并且此顺序保留各个执行流中定义的指令顺序。 我们将在本节末尾讨论放松这一假设的含义，并且在调查中我们考虑的方法与此假设无关。

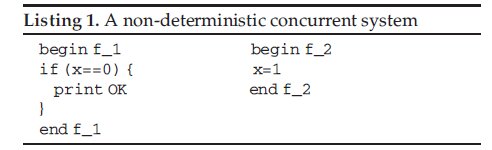
在顺序一致性的假设下，我们可以在具体程序执行中利用历史来模拟多个执行流的指令的交织，该历史是不同执行流的有序指令序列。

在共享存储器系统中，历史包括对数据项的读和写操作的调用序列。 由于通常对共享数据项的操作不是原子操作，因此我们将操作op的调用和终止建模为两个不同的指令。 如果在o的调用和终止之间发生o’的调用，则操作o’的执行与另一操作o的执行重叠。 在消息传递系统中，历史包括原子发送和接收操作的序列。

给定并发系统S的具体执行ex，历史Hex是一系列指令，其中（i）包含所有且仅包含ex的各个执行流的指令的并集，以及（ii）保留所述指令的顺序。 单独的执行流程：对于在ex中发生并且属于相同执行流程f的所有指令oi和oj，如果oi在f之前的oj之前发生，则oi在oj之前出现在Hex中。

我们使用术语交错执行ex来指示Hex中定义的指令的顺序。

清单1举例说明了来自多个执行流的指令交错对执行结果的影响。 在清单1中，执行流f1和f2都访问具有初始值0的公共数据项x，f2将1写入x，而如果f1读取为x，则f1打印OK。 多个交错是可能的。 如果写入操作x = 1的f2发生在历史记录中f1的读取操作x== 0之前，则f1读取1并打印OK，否则f1读取0并且不打印任何内容。



不假设顺序一致性的系统指的是放松的内存模型。 引用宽松模型的系统的示例是共享存储器系统，其中由于多核架构中的多个核的高速缓存之间的惰性同步，不同的执行流可以观察到不同的操作顺序。 一些流行的编程语言通过允许编译器在单个执行流程中重新排序操作来提高性能来引用宽松的模型。 这在Java [8]，[9]和C ++ [10]的内存模型中是允许的。

宽松系统的历史可能违反了表征顺序一致系统历史的属性，导致在顺序一致性假设下不会发生的新类型的可能并发故障。在本调查中，我们回顾了在顺序一致性下工作的许多方法。 假设和少数几种扩展到松弛模型的方法。

2.3同步机制

并发编程语言提供各种同步机制来约束指令的顺序，从而防止错误的行为。 同步机制取决于并发范例，同步结构的粒度以及对系统体系结构施加的约束。 例如，Java提供同步块和原子指令，以确保在没有其他执行流的指令交错的情况下执行程序块[11]。 其他编程语言（如C和C ++）提供锁，互斥锁和信号量来约束代码区的并发执行。然而，其他并发编程环境，如Posixthreads和OpenMP，提供屏障同步来约束对多个执行流同时执行的代码区的访问：障碍和阶段引入了组中所有执行流必须在允许任何执行流之前到达的程序点 继续[12]。 在消息传递的上下文中，实现基于actor的范例的编程语言和库确保以原子方式单独处理单个消息[13]。 同步消息传递确保消息的发送方只有在消息成功传递给接收方后才能进行[14]。

2.4测试并发系统

在本文中，我们将重点放在针对并发故障的软件测试技术上，这些故障是由意外执行流程的指令的意外交错引起的。 并发故障可能非常难以显示和重现，因为它们仅在存在可能很少执行的特定交错时才显示。 为了揭示并发性错误，并发系统的测试技术不仅需要对可能无限的输入空间进行采样，还需要对可能的交叉空间进行采样，这可能会随着执行流的数量和组成流的指令数量呈指数增长。

到目前为止，已经提出的用于测试并发系统的许多方法解决了问题的不同方面。 我们对文献的详细分析产生了一个简单的概念框架，该框架捕获了问题的不同方面，并涉及测试并发系统的许多方法。 图1展示了我们用于提供问题综合视图和组织此调查的概念框架。

测试并发系统的方法处理特定类型的目标系统，并解决图1中用矩形可视化的问题的三个主要方面中的一个或多个：生成测试用例，选择交错并将结果与oracles进行比较。 生成测试用例相当于对程序输入空间进行采样，并生成一组有限的测试用例来运行目标系统。 选择交织量相当于利用执行流的不同交织来增加测试用例，以执行以不同顺序处理相同输入数据的操作。 将结果与oracles进行比较相当于检查目标系统相对于某些oracles的行为。 我们在文献中发现的方法集中在生成测试用例或选择交错，有时也处理与oracles的比较。

图1给出了测试技术的概念框架，但没有规定具体的过程。 一些方法可以首先生成一组测试用例和一组相关的交织，然后将执行结果与oracles进行比较，而其他方法可以通过在识别后立即执行每个交织来交替交织的选择和与oracle的比较。

生成测试用例的方法通过考虑目标系统对输入空间进行采样以生成一组有限的测试用例。 它们还可选地考虑交织的目标属性，提供关于目标系统的附加信息的系统模型，或两者。

用于选择交织的方法识别要执行的相关交织的子集，并且将交织空间作为整体或交织的一些特定属性作为目标。 整体上以交织空间为目标的技术，此后的空间探索技术，随机地，详尽地或通过一些覆盖标准或启发法来探索交织的空间。 两类相关的太空探索技术是压力测试和有界搜索技术。

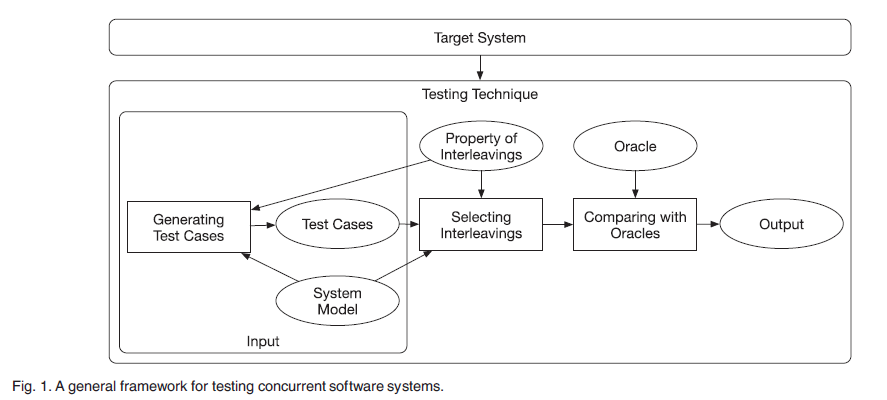
正如我们在第4节中详细讨论的那样，交错的属性通常识别可能暴露并发故障的执行流之间的交互模式。针对一些交织属性的方法，此后基于属性的技术，旨在识别最可能暴露这种模式的交织。

一些基于属性的技术使用交织的属性不仅用于选择交织的相关子集，而且还用于生成可以执行所识别的交织的一组测试用例。这些技术将生成引导到可以表现出暴露感兴趣的属性的交错的测试用例。

大多数基于属性的技术依赖于一些动态或混合分析来构建系统的抽象模型并捕获程序指令之间的顺序关系，然后利用该模型来识别暴露感兴趣的属性的一组交错。

一些基于属性的技术（此后称为检测技术）使用该模型来简单地检测所分析的迹线中感兴趣的图案的存在。其他基于属性的技术（此后称为预测技术）也寻找可能暴露感兴趣的属性的替代交织，通常依赖于模型检查或SAT / SMT求解器。

解决将结果与oracles进行比较的问题的方法在受控环境中执行系统，该受控环境强制所选择的交错并将执行结果与给定的oracle进行比较。



给出了测试技术的概念框架，但没有规定具体的过程。有些方法可能首先生成一组测试用例和一组相关的交错，然后将执行结果与Oracle进行比较，而其他方法则可能通过在确定后立即执行每个交错来交替选择交错，并与Oracle进行比较。

生成测试用例的方法通过考虑目标系统对输入空间进行采样，生成一组有限的测试用例。他们还可以选择考虑交错的目标属性，这是一个提供额外功能的系统模型2。在这个例子中，我们假设x的读和写是原子的。放松原子性假设会产生更多的交错。

有关目标系统或两者的信息。

选择交织的方法确定了要执行的相关交织的子集，并以

图1。测试并发软件系统的通用框架。

作为一个整体的交织空间或交织的某些特定性质。将交织空间作为一个整体来定位的技术，以后的空间探索技术，随机、详尽地探索交织空间，或受某些覆盖标准或启发式方法的驱动。两类相关的空间探索技术是应力测试和有界搜索技术。

正如我们在第4节中详细讨论的，交错的属性通常识别跨执行流的交互模式，这些模式可能暴露并发性错误。针对一些交错属性的方法（以下称为基于属性的技术）旨在识别最有可能公开此类模式的交错。

一些基于属性的技术不仅使用交错的属性来选择相关的交错子集，还可以生成一组可以执行已识别的交错的测试用例。这样的技术引导这一代朝着测试用例前进，这些测试用例可以显示出暴露所关注属性的交错。

大多数基于属性的技术依赖于一些动态或混合分析来构建系统的抽象模型，捕获程序指令之间的顺序关系，然后利用该模型识别出一组暴露所关注属性的交错。

一些基于属性的技术（以下称为检测技术）使用该模型来简单地检测分析跟踪中感兴趣的模式的存在。其他基于属性的技术，以后的预测技术，也会寻找可能暴露感兴趣的属性的可选交错，通常依赖于模型检查或SAT/SMT解算器。

还解决了将结果与Oracle进行比较的问题的方法：在一个受控环境中执行系统，该环境强制选定的交错，并将执行结果与给定的Oracle进行比较。

3并行系统测试研究趋势

在本节中，我们分析了过去15年中对并发系统进行测试的研究，并参考了过去40年的开创性工作。

自60年代初以来，研究并发系统模型的先驱工作，如Karl Adam Petri[15]的研究，以及Tony Hoare[16]和Robin Milner[17]在过程代数方面的鼓舞人心的工作。

70年代，随着分布式体系结构的出现，研究的重点扩展到了对分布式系统的分析和验证，Lipton关于约简理论的影响性工作[18]和Lamport关于分布式系统的开创性工作[7]。

90年代，文献[19]、[20]、[21]中引入了术语“测试具有连续性的并发系统”，并且出现了分析技术，这些分析技术是许多流行的并行系统测试方法的核心[22]、[23]、[24]、[25]、[26]。

在过去15年中，随着多核技术、分布式、网络和移动架构以及新的并发模式的迅速传播，对并发系统测试的研究变得异常专横。我们的调查表明，过去15年中开发的大多数并行软件测试技术都以共享内存系统为目标，只有很少的技术能够处理（分布式）消息传递系统，这些系统主要通过运行时监控和基于模型的验证方法来解决。

为了全面调查并行软件系统测试的新趋势，我们系统地回顾了2000年至2015年的文献：（i）我们搜索了主要科学出版商的在线存储库，包括IEEE Explore、ACM Digital Library、Springer Online Library和Elsevier Online。图书馆，更普遍的是通过流行的在线搜索引擎，如谷歌学者和微软学术搜索，我们收集了2000年发表的论文，这些论文在标题或摘要中呈现了以下一组关键词：“测试+并发”、“测试+多线程”、“测试+并行”“，”testing+distributed“，（ii）考虑所有引用或引用我们存储库中的论文且符合相同标准的出版物，（iii）手动分析我们存储库中的论文出现的会议和期刊的会议记录，（iv）筛选出我们分析范围之外的论文，如定义的那样。

图2。从2000年到2015年，见证新的研究贡献并处理不同并发属性的出版物数量。

例如，在第2节中，关于硬件测试的论文和关于理论方面或纯监控方法的论文（没有直接应用于软件测试）（参见第7节了解这些论文的概述），以及（v）过滤出研讨会论文和前期工作，这些论文后来被会议或期刊出版物收录。

我们选择了过去15年发表的94篇论文，并确定这些论文是相关出版物集群的独特代表。本文主要研究了并发性的主要特性，反映了不同社区对并发系统测试的兴趣。我们将在第6.8节中讨论我们的选择可能存在的偏差。

图2和3分别根据论文所涉及的并发属性和所涉及的研究群体，展示了论文在过去几年中的分布情况。这些数字表明了相关的兴趣和结果的增加：超过80%的论文在审议期间（20082015年）的下半年发表，而自2010年以来几乎有65%的论文发表。图2表明，最近的研究集中于特定的交织特性，我们在第4节中使用这些特性作为主要的分类标准。图3表明，人们对软件工程和编程语言社区的兴趣与日俱增，对系统和正式方法社区的兴趣与日俱增，这些社区的主要关注点分别是监测并发系统和理论调查的技术。我们将在相关工作第7节中简要讨论这两个主题，因为它们是相关的，但不是本文主题的核心。我们对文献的分析导致了第2节图1框架的定义，这启发了本节提出的分类模式，并在第5节中用于组织我们的调查。图4概述了我们的分类模式，其中包括七种不同的分类标准，它们根据以下内容区分技术：

输入。大多数方法假设一组输入测试用例的可用性，很少有方法在被测试系统的某些模型上工作，而其他方法则同时需要测试用例和模型。

三。我们通过出版场所确定社区。

图3。2000年至2015年在不同研究社区见证新研究贡献的出版物数量。

交错的选择。许多方法都涉及到交错的一些属性来选择相关的子集（基于属性），其他方法要么详尽地探索交错的空间，要么利用一些覆盖标准或启发式（空间探索）。基于属性的检测技术只需检查执行跟踪中是否存在感兴趣的模式，而预测技术还选择可能暴露感兴趣的属性的新交错。

交错的属性。许多方法选择引用某些特定并发属性的交错：数据争用、原子性冲突、死锁、组合或顺序冲突属性。

输出和Oracle。有些技术只是报告所探索的交错是否满足感兴趣的属性（满足交错的属性），而另一些技术则根据某些Oracle以系统崩溃、死锁或违反断言的形式识别失败的执行。

保证。不同的技术在其结果的精确性和正确性方面提供了不同程度的保证。

目标系统。大多数技术针对的是一些特定类型的系统，这些系统依赖于通信模型（共享内存、消息传递或通用，即独立于通信范式）、编程范式（通用或特定，如面向对象、基于参与者或基于事件的范式）和一致性模型（后续或放松）。

测试技术。这些技术在分析类型（可以是动态的或混合的）、测试类型（可以是功能性的或非功能性的）、测试粒度（单元、集成或系统测试）、测试范围以及用于实现该技术的测试架构（可以是CEN）方面有所不同。分散的或分布的。

在本节的其余部分中，我们将详细讨论分类标准。

4.2交错的选择

选择交错是许多方法的主要目标。一些技术利用感兴趣的属性来选择交错：我们将它们称为基于属性的。其他技术是彻底地或随机地探索交错的空间，可能利用启发式或覆盖标准：我们称之为空间探索技术。

4.2.1基于属性的技术

基于属性的技术根据感兴趣的一个或多个属性选择交错。它们通常对执行跟踪应用某种形式的分析，以确定指令之间的相关同步约束。例如，锁集分析侧重于基于锁的同步，并查找对不受锁保护的共享数据项的访问[23]。happensbefore分析通过捕获一般的顺序约束扩展了这种方法。分析前发生的不同类型适用于不同的同步机制[28]，并呈现出各种成本准确性权衡[29]，[30]。

基于属性的技术利用通过分析识别的顺序信息来简单地理解所分析的跟踪是公开感兴趣的属性（检测技术），还是识别可以公开感兴趣的属性的可选交错（预测技术）。

4.2.2空间探测技术

空间探索技术在不涉及特定属性的情况下探索交错空间，包括：

压力测试。压力测试方法多次执行测试套件，目的是观察不同的交错。它们不提供观察交织空间给定部分的任何保证，也不引入任何机制来提高执行新交织的概率。

彻底的探索。详尽的探索方法旨在执行所有可能的交错。由于一般情况下交错空间可能很大，这些技术要么限制输入测试用例的指令数量和执行流，要么引入探索空间的边界[31]。它们还经常采用诸如动态部分阶约简[32]之类的约简技术，以避免执行等效交错。

覆盖标准。覆盖标准根据探索空间的深度确定了要练习的交错部分。例如，在共享内存系统中，覆盖标准可能要求对于属于不同执行流并在同一数据项d上操作的每对指令i1和i2，至少应存在一个测试用例，该测试用例执行i1在i2之前发生的交织，以及执行i2之前发生的交织。在I1之前发生。

试探法。启发式方法指导对交错空间的探索。例如，一些技术通过它们相对于执行的技术的多样性来优先考虑交错。类似地，一些其他的技术优先考虑可以通过引入有限数量的调度约束来获得的交错。

4.3交织物的性质

基于属性的技术针对交错的特定属性，这是执行流之间的交互模式，可能违反开发人员对指令执行顺序的假设。一些基于属性的技术针对交错的经典属性：数据争用、原子性冲突和死锁。其他技术结合了多种经典特性。然而，其他技术集中在特定于程序或特定于域的顺序冲突上。

4.3.1数据竞赛

当来自不同执行流的两个操作同时访问同一个数据项d时，至少有一个操作是写操作，并且没有使用同步机制来控制对d的（顺序）访问时，就会发生数据争用。如果在执行期间不能发生数据争用，则系统是无数据争用的。

清单2显示了数据竞争的一个例子。对于string类型的数据项x，执行流f\_1和f\_2都会写入。让我们假设底层编程语言可以原子地写入32位（4个字符）。如果f\_1和f\_2同时在x上开始写入，x的结果值将

“aaaa”和“bbbb”的任意组合，例如x='aabbbb'，其中前4个字符来自f\_1中的写入指令，其余字符来自f\_2中的写入指令。

清单2。数据竞赛示例

开始信息1开始信息2

x='aaaaaa'x='bbbbbbb'

Enff1 Enff2

数据竞争可能表示违反了对单个操作执行的原子性假设。这样的违反破坏了系统行为的可序列化性。可序列化的概念最初是在数据库上下文中定义的。

系统作为交易正确性的保证[33]。在我们的上下文中，如果一个历史等价于一个序列历史，那么它就是一个满足所有原子性假设的历史。如果两个历史记录为所有数据项生成相同的值，则它们是等效的。4数据竞赛意味着对数据项的不受控制的访问，这可能是错误，也可能不是错误。例如，在清单2的示例中，值x='aaaabbb'可能是有效的，或者不取决于应用程序逻辑。

数据竞赛技术以低水平或高水平的数据竞赛为目标，提出了不同类型的分析和算法。检测低级数据竞争的技术在细粒度级别上工作，通常考虑对单个内存位置的访问。检测高级数据争用的技术检查由于在共享的复杂数据结构上调用并发操作的不同执行流导致的错误行为，例如面向对象程序中对象或库的公共方法。

当处理两条消息的代码片段访问一个公共数据项，并且至少有一个片段修改该数据项时，实现消息传递范式的程序中也可能发生数据竞争。

4.3.2违反原子性

原子性冲突将原子性的概念扩展到操作序列。当假定以原子方式执行的执行流的一系列操作与来自其他流的冲突操作交错时，就会发生原子性冲突。许多原子性冲突技术使用序列化来验证交织的正确性。但是，检查交织的可序列化性可能非常昂贵，因为它需要将交织与所有可能的串行历史进行比较。因此，许多技术经常检查已知不可序列化的交织的特定模式。

清单3显示了原子性冲突的一个例子。在这个例子中，我们假设x最初持有一个非负值，并且我们期望它的值始终保持非负值。假设F\_1是原子执行的，即F\_1中的操作序列是原子执行的，而没有F\_2的交叉操作。但是，如果没有通过同步机制正确地强制f\_1的原子性，f\_2中的冲突操作x\_可能在f\_1的两个操作之间发生，导致x的值变为负值（1）。

清单3。原子性侵犯的一个例子

开始信息1开始信息2

如果（x＞0）x＝0

x=x-1端f\_1端f\_2

在实现消息传递范式的程序中，当处理两个或多个应以原子方式执行的消息的代码片段被一些其他消息的处理交错时，也可能发生原子性冲突。

根据原子块的定义，我们区分了两类主要的检测原子性冲突的方法，即以代码为中心和以数据为中心的方法。

4。在文献中，这种属性被称为视图可序列化，与冲突可序列化相反[34]。

以代码为中心。以代码为中心的技术在粗粒度级别上工作。它们针对应根据系统的某些规范原子地执行的代码块，并验证系统实现是否保证原子性要求。原子代码块可以是显式指定的[27]、[35]或隐式假定的[36]、[37]，基于一些启发式或编程范式的一些假设。

以数据为中心。Vaziri等人于2006年首次研究了以数据为中心的原子性违规。世卫组织引入了原子集可序列化的概念，将其作为确保数据一致性的编程抽象[38]。原子集的可序列化性建立在原子集的概念之上，原子集表示与某些一致性约束相关的数据项集，以及用于更新原子集中变量的代码块的工作单元。在原子集编程模型中，开发人员只需要指定原子集和工作单元，编译器推断同步机制以避免潜在的危险交叉。

一些测试技术使用违反原子集序列化的模式来选择危险的交错。由于原子集可序列化方法需要定义原子集，因此这些技术要么依赖于某些代码注释，要么使用启发式方法或对正在使用的编程范式的假设来推断原子集。

4.3.3死锁

当同步机制无限期地阻止某些执行流继续执行时，就会发生死锁。这种情况发生在循环等待的情况下，即给定流集的每个执行流都在等待同一流集中的另一个执行流进行，因此无法继续其自身的执行。

清单4显示了由于不正确地使用锁而导致死锁的一个简单示例。锁提供了两个原子原语，acquire（l）和release（l）。当执行流f获取锁l（acquire（l））时，其他执行流f0在f释放锁（release（l））之前不能获取l。锁用于实现互斥：例如，为了保证一组操作的原子性，每个执行流在O中访问一个操作之前都要获取一个锁L，在终止对资源的访问时释放锁L，以防止其他流同时在O中执行一个操作。

清单4。死锁的例子

开始信息1开始信息2

获取（l1）获取（l2）

获取（l2）获取（l1）

……

释放（l2）释放（l1）

释放（l1）释放（l2）

Enff1 Enff2

在清单4的示例中，执行流f\_1在释放l1之前获取第一个锁l1，然后锁定l2，而执行流f\_2在释放l1之前获取锁l2。如果f\_1在f\_1前进之前获得l1，f\_2在f\_1前进之前获得l2，则f\_1被阻止。

等待f\_2释放l2和f\_2被阻止等待f\_1释放l1，从而导致死锁。

死锁可能取决于对共享资源（资源死锁）的访问顺序不正确，或者取决于执行流之间的通信协议不正确（通信死锁）[39]。

资源死锁。当一组执行流试图访问某些公共资源时，会发生资源死锁，并且集合中的每个执行流都请求由集合中的另一个执行流持有的资源。清单4中的代码是资源死锁的一个例子。

通信死锁。当一组执行流交换消息时，消息传递系统中会发生通信死锁，其中每个消息都等待来自同一组中另一个执行流的消息。

4.3.4组合

一些测试技术处理交织的组合属性，这是上面定义的多个属性之一。

4.3.5违反订单

数据竞争、原子性冲突和死锁是经典的、研究得很好的属性，通常会导致特定的交叉顺序冲突。有些技术针对的是程序或特定于域的交错顺序的冲突，这些冲突无法追溯到经典的交错属性。我们将这些技术称为违反顺序的方法。例如，一些技术以并发面向对象程序的领域为目标，并着重于导致空指针取消引用的交错，当一个执行流在另一个执行流[40]将对象引用设置为空后错误地试图取消引用该对象引用时，会发生这种情况，即使使用类数据竞赛。

4.4输出和Oracle

测试技术产生两种类型的输出，一些简单地产生满足交织的属性，这些交织是公开感兴趣的属性的交织，而另一些则比较执行交织与Oracle产生的结果，并返回失败的执行，从而实现图1中的“与Oracle比较”功能。

一般来说，并非所有表现出利益性质的交错都会导致失败。因此，满足交织的输出特性的技术可能导致误报。例如，一些检测数据竞争的技术可能会发出许多良性数据竞争的信号。

oracle定义了区分可接受执行和失败执行的标准，可以是系统崩溃、死锁或违反断言oracle。系统崩溃和死锁Oracle，也称为隐式Oracle，分别标识导致系统崩溃和死锁的执行。违反断言oracle利用关于系统正确行为的断言，基于对编程范式的显式规范或隐式假设。

4.5包

选择交错的不同技术保证了结果的不同级别的有效性。如果一种技术只产生可以在某些具体执行中观察到的交错，那么它就保证了结果的可行性。并不是所有的技术都能保证交织的可行性，例如，一些分析轨迹并产生观测操作的可选交织的预测技术可能会遗漏一些程序约束，从而返回不符合任何可行执行的交织。

如果一种技术只产生导致Oracle违规的交错，那么它就保证了结果的可靠性。如果一种技术产生了在某些具体执行中可以观察到的所有交错，并导致Oracle违规，那么它就保证了结果的完整性。如果一种技术只识别出可行的交叉点，而这些交叉点展示了所考虑的测试用例所感兴趣的特性，那么它就保证了属性的可靠性。如果一种技术能够识别出所有可行的交错，从而展示出所考虑的测试用例所感兴趣的属性，那么它就保证了属性的完整性。

属性稳健性和属性完整性的概念只适用于基于属性的技术。一些基于属性的技术的作者将他们的方法描述为健全的和/或完整的，关于他们考虑的交错的属性。然而，他们的说法往往依赖于仅使用某些特定同步机制的假设。在一般情况下，在大多数基于属性的技术中采用的顺序关系和分析的类型，如发生在分析之前[28]或因果关系之前[29]，可以引入妨碍可靠性和完整性的近似值[30]。

完整性和完整性描述了检测并发故障的技术的准确性。属性稳定性和属性完整性描述了检测公开给定属性的交错的技术的准确性。

4.6目标系统

我们识别出了三个描述目标并发系统类型的元素：通信模型、编程范式和一致性模型。

通信模型指定执行流如何相互作用，包括共享内存和消息传递模型。通用技术针对两种类型的系统。

许多技术针对特定的编程范式，有时由目标同步机制识别。例如，一些技术利用面向对象范式的特定属性，例如状态或子类型可替换性的封装。类似地，其他技术建立在基于参与者的系统中提供的假设之上。有些技术只考虑使用特定同步机制时产生的故障，例如由于不正确使用基于锁的同步而导致的死锁。然而，其他技术并没有对所采用的编程范式做出任何假设，也没有与通用系统一起工作。

最后，一些测试技术假设一个顺序一致性模型，而其他技术可以应用到宽松的一致性模型。

4.7测试技术

我们描述了迄今为止沿着五个轴提出的许多测试技术，它们实现的分析类型，类型，粒度和范围的测试技术，测试和建筑类型收养他们。

4.7.1分析

测试技术无论实施或混合的动态分析。动态分析技术的使用源于executions只读信息系统下的测试。混合分析技术使用两个静态和动态信息。严格的说，技术rely静态信息，在线测试技术是不是唯一的，和因此而在本调查的范围。我们discuss最相关的静态分析技术的相关工作（第7节）。

4.7.2式测试

测试技术的可能的目标或功能或非功能属性。功能测试技术的正确性检查的结果与程序有关的一个给定的预言，但非功能性测试验证技术nonfunctional置业如反应时间和scalability。

4.7.3粒度测试

不同的测试技术工作的各种粒度的水平，跨度从两个单元和系统集成。单元测试技术在孤立的个体单位的目标。例如，在面向对象程序的单元测试模式，认为在没有隔离带，为他们的客户互动之与其他部件的系统。集成测试技术在福克斯互动之间的单位，和检查之间的通讯接口测试厂为单位的预期。系统测试技术指标的系统作为一个整体，和验证，这符合其要求。在第5节中，我们都表示的粒度级别和类型的系统部件的试验技术的考虑。

4.7.4范围测试

一些测试技术有一个有限的范围的发展过程。例如，许多技术是用于系统验证，当其他技术指标的回归测试。

4.7.5测试体系结构

指两种不同的测试技术，测试architectures characterize《混凝土基础设施使用的运动的系统的测试。这样的基础设施是组合的一个或多个测试驱动程序，这两种产品的输入数据的一个或多个执行流的系统下测试，这就是要守的输出产生的系统下测试。例如，一个测试客户端服务器的分布式操作系统，CAN（initialize（客户端提交的要求，有两个服务器，等为replies从服务器收到和评估未来replies登个Oracle。

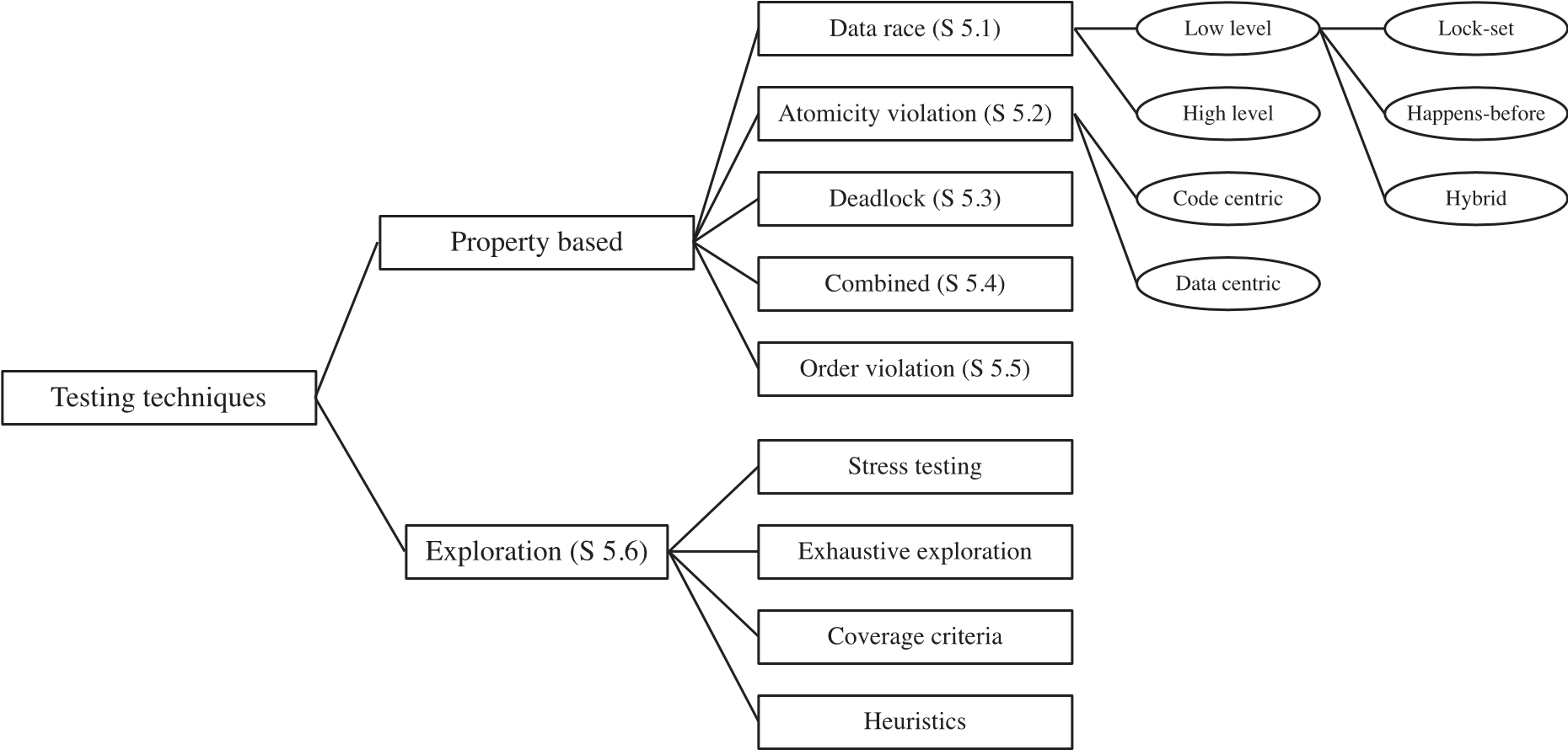
试验architectures或者可以集中，当一个单一的互动管理局与驱动系统下测试，或分布，当超过一个驱动与互动管理局（不同的）的执行流下的系统测试。在以上的情况下，一个客户端服务器的分布式系统的体系结构exemplifies集中测试。一种测试A的P2P系统的分布式测试环境与经营为每个节点是一个简单的实例的分布式测试架构。

图5。详细调查的组织。

我们区分了同步和独立的分布式测试体系结构。在同步测试体系结构中，驱动程序通过交换消息来相互协调，而在独立测试体系结构中，驱动程序独立执行，不交换消息来协调它们的操作。在分布式测试体系结构中，驱动程序同步被用来克服可控性和可观测性问题。如果驱动程序不能分别确定何时产生特定的输入，或者特定的输出是否是响应特定的输入而产生的，就会出现这些问题[41]。

5详细调查

根据第4节讨论的标准，我们对测试并发系统的主要方法进行了分类。图5显示了本节的组织结构。本文以交错体的选择为主要分类标准，对物性勘探技术和空间勘探技术进行了区分。我们根据交织的目标属性将基于属性的方法分类为数据竞争（第5.1节）、原子性冲突（第5.2节）、死锁（第5.3节）、组合（第5.4节）和顺序冲突（第5.5节），并讨论第5.6节中的空间探索技术。

该组织将实现相关方法和算法类的方法组合在一起，以检测故障。这是一种基于属性的技术，它利用交错的相同属性，通常针对相同类型的并发故障，以及空间探索方法，通常针对一般类型的故障。

我们根据标准将这些技术的分类汇总在六个表中。表1、2、3、4和5概述了基于属性的技术。表6空间探索技术概述。这六张桌子结构相同。行指示技术及其名称（如果可用）或提议该技术的论文作者的名称。行按子类别分组（如果适用），并报告按相同子类别中的主要贡献排序的方法。大多数方法的贡献是多方面的，我们根据我们认为的核心新颖性列出方法，并在特别相关的时候提到其他元素。这些列对应于图4中确定的标准。

在这六个表中，我们没有明确报告测试类型、测试范围和测试体系结构，因为我们分析的所有技术（i）实现功能测试，唯一的例外是侧重于性能的speedgun，（i i）属于验证测试的范围，但recontest、simrt和speedgun除外，后者是e显式设计用于回归测试，以及（iii）实现一个集中的体系结构。在表6中，我们也省略了列预测、属性完整性和属性可靠性，因为它们只适用于基于属性的技术。

5.1基于属性：数据竞赛

我们分类大量的技术设计，以检测数据竞争在共享内存程序，根据其粒度和类型的分析，他们执行。我们考虑的技术，目标低级别和高级别的数据竞争，我们进一步分类低级别技术作为锁集，发生之前和混合分析技术。

低水平的数据竞争检测技术。低级数据竞争检测技术针对的是在单个内存位置级别上发生的数据竞争。它们依赖某种形式的分析来跟踪给定执行跟踪中内存指令之间的顺序关系，并检测数据争用的发生，或者预测在可选的交错中是否可以进行数据争用。

这些技术要么依赖于锁集分析，它只标识不受锁保护的并发内存访问，要么依赖于在分析之前发生的操作，它检测并发内存访问之间的某些顺序关系。依赖于在分析之前发生的测试技术通常声称是属性完整的，这意味着它们可以检测所有可以用输入执行跟踪的可选计划生成的数据争用。但是，在分析之前通常是保守的：例如，当它在执行跟踪中观察到一个锁的释放，然后获取相同的锁时，它将这两个操作解释为

表1

数据竞争检测技术

sm共享内存seq序列一致性s系统测试mp消息传递rel松弛一致性u单元测试h混合分析d动态分析完全有序，但它们可能以不同的顺序出现在同一痕迹的其他交错中。因此，传统的先验分析可能会遗漏一些可能的交错，从而遗漏一些错误[29]。

属性完整性问题已经通过实现关系之前发生的变量来解决，这些变量更准确地捕获事件的顺序，从而减少遗漏某些错误的可能性[29]、[30]，或者通过利用模型检查来探索根据overre不允许的指令重新排序。严格是在分析之前发生的，但在实践中仍然是可能的。一些混合解决方案结合了锁具分析不准确但成本低廉的优点，以及分析前更准确但成本更高的优点[42]。

高级数据竞态检测技术。高级数据竞争检测技术针对复杂的数据结构，如面向对象程序中的对象，并寻找导致结果不兼容的交错。

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | TestCase | Model |  | Sat.Interl. | Crash  Deadlock  Assert.Viol. | Predict.  Commun.  Paradigm | | | | | | Consist. |  | Granularity | Analysis |  | Prop.Complet. | Prop.Sound. | Complet. | Soundness | Feasibility |
|  |  |  |  |  |  | Low level — Lockset | | | | | |  |  |  |  |  |  |  |  |  |  |
| Shacham et al. | @ |  |  |  | @ @ @ | @ SM General | | | | | | Seq |  | S | D |  | @ |  | @ | @ | @ |
| Racez | @ |  |  | @ |  | SM General | | | | | | Seq |  | S | D |  |  |  | - | - | @ |
| von Praun and Gross | @ |  |  | @ |  | SM OO | | | | | | Seq |  | S | H |  |  |  | - | - | @ |
| ACCORD | @ | @ | @ | |  | @ SM Fork-join | | | | | | Seq |  | S | D |  |  |  | - | - |  |
|  |  |  |  |  |  | Low level — Happens-before | | | | | |  |  |  |  |  |  |  |  |  |  |
| FastTrack | @ |  |  | @ |  |  | |  | SM | General | | Seq |  | S | D |  |  |  | - | - | @ |
| LiteRace | @ |  |  | @ |  |  | |  | SM | General | | Seq |  | U | D |  |  |  | - | - | @ |
| Pacer | @ |  |  | @ |  |  | |  | SM | General | | Seq |  | S | D |  |  |  | - | - | @ |
| SOS | @ |  |  | @ |  |  | |  | SM | General | | Seq |  | S | D |  |  |  | - | - | @ |
| Carisma | @ |  |  | @ |  |  | |  | SM | General | | Seq |  | S | D |  |  |  | - | - | @ |
| ReEnact | @ | @ | @ | |  |  | |  | SM | General | | Seq |  | S | D |  |  |  | - | - | @ |
| Narayanasamy et al. | @ |  |  | @ |  | @ | |  | SM | General | | Seq |  | S | D |  |  | @ | - | - | @ |
| Frost | @ |  |  |  | @ @ @ | @ | |  | SM | General | | Seq |  | S | D |  |  |  |  |  | @ |
| Portend | @ |  |  | @ |  | @ | |  | SM | General | | Seq |  | S | D |  |  | @ | - | - | @ |
| Tian et al. | @ |  |  | @ |  |  | |  | SM | General | | Seq |  | S | D |  |  |  | - | - | @ |
| RDIT | @ |  |  | @ |  | @ | |  | SM | General | | Seq |  | S | D |  |  |  | - | - |  |
| Smaragdakis et al. | @ |  |  | @ |  | @ | |  | SM | General | | Seq |  | S | D |  |  |  | - | - |  |
| DrFinder | @ |  |  | @ |  | @ | |  | SM | General | | Seq |  | S | D |  |  |  | - | - | @ |
| RVPredict | @ |  |  | @ |  | @ | |  | SM | General | | Seq |  | S | D |  |  |  | - | - |  |
| WebRacer | @ |  |  | @ |  |  | |  | SM | Web platforms | | Seq |  | S | D |  |  |  | - | - | @ |
| EventRacer | @ |  |  | @ |  |  | |  | SM | Event-based | | Seq |  | S | D |  |  |  | - | - | @ |
| DroidRacer | @ |  |  | @ |  |  | |  | SM | Android apps | | Seq |  | S | D |  |  |  | - | - | @ |
| Java RaceFinder | @ |  |  | @ |  | @ | |  | SM | General | | Rel |  | S | D |  | @ |  | - | - | @ |
| Relaxer | @ |  |  | @ |  | @ | |  | SM | General | | Rel |  | S | D |  |  |  | - | - | @ |
|  |  |  |  |  |  | Low level — Hybrid | | | | | |  |  |  |  |  |  |  |  |  |  |
| Choi et al. | @ |  |  | @ |  |  | SM | | | | General | Seq |  | S | H |  |  |  | - | - | @ |
| Wester et al. | @ |  |  | @ |  |  | SM | | | | General | Seq |  | S | D |  |  |  | - | - | @ |
| RaceMob | @ |  |  | @ |  |  | SM | | | | General | Seq |  | S | H |  |  | @ | - | - | @ |
| RaceFuzzer | @ |  |  |  | @ | @ | SM | | | | General | Seq |  | S | D |  |  |  |  | @ @ | |
| RaceTrack | @ |  |  | @ |  |  | SM | | | | General | Seq |  | S | D |  |  |  | - | - @ | |
| Goldilocks | @ |  |  | @ |  |  | SM | | | | General | Seq |  | S | H |  |  |  | - | - @ | |
| MultiRace | @ |  |  | @ |  |  | SM | | | | General | Rel |  | S | D |  |  |  | - | - @ | |
| SimRT | @ |  |  |  | @ | @ | SM | | | | General | Seq |  | S | D |  |  |  |  | @ @ | |
| Racageddon |  |  |  | @ |  | @ | SM | | | | General | Seq |  | S | D |  |  |  | - | - @ | |
| Narada | @ |  |  | @ |  | @ | SM | | | | General | Seq |  | S | D |  |  |  |  | @ @ | |
|  |  | |  |  |  |  | High level | | | |  |  |  |  |  |  |  |  |  |  | |
| Colt | @ @ @ | | | |  | @ | SM | | | | OO | Seq |  | U | D |  |  |  | - | - @ | |
| Dimitrov et al. | @ @ @ | | | |  |  | SM | | | | OO | Seq |  | U | D |  |  |  | - | - @ | |

在这张表格和论文中的所有表格中，这些技术根据其在每个子类别中的出版日期进行分类，并按照提议者的指示命名，或者在没有名字的情况下由论文的作者命名。图例（所有表通用）：TABLE 2

Atomicity Violation Detection Techniques

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Input  TestCase | Model | Sat.Interl. | Output / Oracle  Crash | Deadlock | Assert.Viol. | Select.  Interl.  Predict.  Commun. | Target System  Paradigm | Consist. | Testing Tech.  Granularity | Analysis |  |  | Guarantees | |  |
| Prop.Complet. | Prop.Sound. | Complet. | Soundness | Feasibility |
|  |  |  |  |  |  |  | Code centric |  |  |  |  |  |  |  |  |  |
| Atomizer | @ | @ | @ |  |  |  | SM | General | Seq | S | D |  |  | - | - | @ |
| SVD | @ |  | @ |  |  |  | SM | General | Seq | S | D |  |  | - | - | @ |
| AVIO | @ |  | @ |  |  |  | SM | General | Seq | S | D |  |  | - | - | @ |
| Velodrome | @ |  | @ |  |  |  | SM | General | Seq | S | D |  |  | - | - | @ |
| AtomFuzzer | @ |  |  | @ | @ |  | @ SM | General | Seq | S | D |  |  |  | @ | @ |
| HAVE | @ |  | @ |  |  |  | @ SM | General | Seq | S | H |  |  | - | - |  |
| Penelope | @ |  |  | @ |  | @ | @ SM | General | Seq | U | D |  |  |  | @ | @ |
| Wang and Stoller | @ |  | @ |  |  |  | @ SM | OO | Seq | S | D | @ |  | - | - |  |
| CTrigger | @ |  |  | @ | @ | @ | @ SM | General | Seq | S | D |  |  |  | @ | @ |
| Falcon | @ |  | @ |  |  |  | SM | General | Seq | U | D |  |  | - | - | @ |
| Best | @ |  | @ |  |  |  | @ SM | General | Seq | S | H |  |  | - | - | @ |
| DoubleChecker | @ | @ | @ |  |  |  | SM | General | Seq | U | D |  |  | - | - | @ |
| Intruder | @ |  | n.a. | n.a. | n.a. | n.a. | SM | General | Seq | U | D | n.a. | n.a. | n.a. | n.a. | n.a. |
|  |  |  |  |  |  |  | Data centric |  |  |  |  |  |  |  |  |  |
| Muvi | @ |  | n.a. | n.a. | n.a. | n.a. | n.a. SM | General | Seq | S | D | n.a. | n.a. | n.a. | n.a. | n.a. |
| Hammer et al. | @ |  | @ |  |  |  | @ SM | General | Seq | S | H |  |  | - | - | @ |
| AssetFuzzer | @ | @ | @ |  |  |  | @ SM | General | Seq | S | D |  |  | - | - | @ |
| ReConTest | @ |  |  | @ | @ | @ | @ SM | General | Seq | S | D | @ |  | @ | @ | @ |

任何串行执行。检测高级别数据竞争的na€ve方法包括将交错执行的结果与所有可能的串行执行进行比较。对所有可能的串行执行进行详尽分析的直观的可伸缩性问题是由许多测试技术来解决的，这些测试技术利用了开发人员提供的某种形式的规范，这些规范指示了操作之间的相关顺序特征，例如交换性。

5.1.1低位锁具

Savage等人提出了数据竞态检测的锁集分析。在90年代后期，基于李普顿在70年代引入的还原理论[18]。Savage等人的擦除方法[23]既解决了静态数据竞争分析的保守局限性，也解决了分析前发生的性能问题，这是当时调查的两种常见的检测数据竞争的方法。事实上，在90年代，在分析被认为太昂贵之前就已经发生了，因为它需要每个执行流的信息来说明对每个共享数据项的并发访问。锁集分析通过动态计算访问共享数据项时持有的公共锁集，以及标识访问相同共享数据项而不共享任何锁的执行流，揭示了可能的数据竞争。通过只考虑基于锁的同步，锁集分析提高了在分析之前发生的效率，因为它只需要存储关于

表3

死锁检测技术

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Input  TestCase | Model | Sat.Interl. | Output / Oracle  Crash | Deadlock | Assert.Viol. | Select. Interl.  Predict. | Commun. | Target  System  Paradigm | Consist. | Testing Tech.  Granularity | Analysis |  | Guarantees | | |  |
| Prop.Complet. | Prop.Sound. | Complet. | Soundness | Feasibility |
| Goodlock | @ |  | @ |  |  |  | @ | SM | Lock sync | Seq | S | D |  |  | - | - |  |
| DeadlockFuzzer | @ |  |  |  | @ |  | @ | SM | Lock sync | Seq | S | D |  | @ |  | @ | @ |
| MagicFuzzer | @ |  |  |  | @ |  | @ | SM | Lock sync | Seq | S | D |  | @ |  | @ | @ |
| Wolf | @ |  |  |  | @ |  | @ | SM | Lock sync | Seq | S | D |  | @ |  | @ | @ |
| ConLock | @ |  |  |  | @ |  | @ | SM | Lock sync | Seq | S | D |  | @ |  | @ | @ |
| Sherlock | @ |  |  |  | @ |  | @ | SM | Lock sync | Seq | S | D |  | @ |  | @ | @ |
| OMEN | @ |  |  |  | @ |  | @ | SM | Lock sync | Seq | S | D |  | @ |  | @ | @ |
| Armus | @ |  |  |  | @ |  |  | SM | Barrier sync | Seq | S | D |  | @ |  | @ | @ |

表4

检测组合属性冲突的技术

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Input  TestCase | Model | Sat.Interl. | Output / Oracle  Crash  Deadlock | Assert.Viol. | Select. Interl.  Predict. | Commun. | Target System  Paradigm | Consist. | Testing Tech.  Granularity | Analysis |  | Guarantees |  |
| Prop.Complet. | Prop.Sound.  Complet.  Soundness | Feasibility |
| Agarwal et al. | @ |  | @ |  |  |  | SM | General | Seq | S | H | | - - | @ |
| Chen and MacDonald | @ |  |  | @ @ | @ | @ | SM | General | Seq | S | H | | @ @ | @ |
| Kahlon and Wang | @ |  | @ |  |  | @ | SM | General | Seq | S | D | | - - |  |
| PECAN | @ | @ | @ |  |  | @ | SM | General | Seq | S | D | | - - | @ |

执行流持有的锁，但由于忽略了由其他同步机制确定的附加顺序关系，因此会降低准确性。因此，基于锁集分析的技术不是性能可靠的。

在过去的15年中，对数据竞争检测的研究主要集中在减少误报的数量，将注意力转移回分析前发生的情况，并转向混合方法。最近仅有的几种专门基于锁集分析的技术，目的是提高精度和效率，或者扩展到新的编程模式。Shacham等人[43]和racez[44]分别提高了精度和准确性，而praun和gross[45]和accord[46]分别将锁集分析扩展到面向对象和基于数组的并发程序。

提高精度。Shacham等人，ppopp 2005[43]。Shacham等人将动态锁集分析与模型检查相结合，检测通用共享内存程序中的数据竞争。它们通过动态锁集分析捕获锁定约束，通过模型检查为执行的跟踪生成可选的交错，并执行由模型检查器识别的交错以显示故障。锁集分析确保了属性的完整性，而不是属性的完整性，而最终的执行阶段则保证了Oracle的完整性和完整性。

提高效率。Racez，ICSE 2011[44]。racez通过只捕获同步操作的一个子集和目标系统执行的内存访问的抽样方法减少了锁集分析的开销，从而以复杂性换取准确性。Racez继承了先前工作（如Literace和Pacer）的采样思想，我们将在本节的其余部分中讨论这一点。Racez通过只检测同步操作并将内存操作检测委托给硬件性能监视单元（PMU）进一步降低了开销。PMU将硬件信息存储在缓冲区中，缓冲区通过系统级调用异步访问，通常用于性能监视。这种用户级仪器（用于监控同步操作）和硬件（用于监控内存访问）的组合导致了非常低的开销，Sheng等人在一些在真实系统上进行的实验中，报告的开销低至2.8%。由于采样引入的近似值，该方法既不具有完整的属性，也不具有良好的属性，因为它依赖于不精确的锁集分析。

表5

检测订单违规的技术

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Input  TestCase | Model | Sat.Interl. | Output /Oracle  Crash | Deadlock | Assert.Viol. | Select. Interl.  Predict. | Commun. | Target System  Paradigm | Consist. | Testing Tech.  Granularity | Analysis |  | Guarantees | | |  |
| Prop.Complet. | Prop.Sound. | Complet. | Soundness | Feasibility |
| PRETEX | @ |  | @ |  |  |  | @ | SM | OO | Seq | U | H |  |  | - | - |  |
| 2ndStrike | @ | @ |  | @ |  |  | @ | SM | OO | Seq | U | D |  | @ |  | @ | @ |
| ConMem | @ |  |  | @ |  |  | @ | SM | General | Seq | S | D |  | @ |  | @ | @ |
| ConSeq | @ |  |  | @ |  | @ | @ | SM | General | Seq | S | H |  | @ |  | @ | @ |
| ExceptioNULL | @ |  |  | @ |  |  | @ | SM | General | Seq | S | D |  | @ |  | @ | @ |
| Maple | @ |  |  | @ | @ | @ | @ | SM | General | Seq | S | D |  | @ |  | @ | @ |
| jPredictor | @ | @ | @ |  |  |  | @ | SM | General | Seq | S | H |  | @ |  | @ | @ |
| Sinha and Wang | @ |  | @ |  |  |  | @ | SM | General | Seq | S | D |  |  | - | - |  |
| DefUse | @ |  | @ |  |  |  |  | SM | General | Seq | S | D |  |  | - | - | @ |
| GPredict | @ | @ | @ |  |  |  | @ | SM | General | Seq | S | D |  |  | - | - |  |
| SimRacer | @ |  |  | @ |  | @ | @ | SM | Process level | Seq | S | D |  | @ |  | @ | @ |
| Cafa | @ |  | @ |  |  |  | @ | MP | Event driven | Seq | S | D |  |  | - | - |  |
| Mutlu et al. | @ |  | @ |  |  |  | @ | MP | Javascript app | Seq | S | D |  |  | - | - |  |

TABLE 6

Techniques for Exploring the Space of Interleavings

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Input  TestCase | | Model | Sat.Interl. | Output / Oracle  Crash | Deadlock | Assert.Viol. | Commun. | Target System  Paradigm | Consist. | Testing Tech.  Granularity | Analysis |  | Guarantees | |
| Complet. | Soundness | Feasibility |
| Stress testing | |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Pike @ | | @ | @ |  |  |  | SM | General | Seq | S | D |  | @ | @ |
| SpeedGun | |  |  |  |  | @ | SM | General | Seq | n.a. | n.a. | @ |  |  |
| Exhaustiveexploration | |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Ballerina |  |  | @ |  |  |  | SM | OO | Seq | U | D |  | @ | @ |
| Sen and Agha |  |  |  | @ | @ |  | MP | General | Seq | S | D |  | @ | @ |
| Basset | @ |  |  | @ | @ | @ | MP | Actors | Seq | S | D | @ | @ | @ |
| CheckMate | @ |  | @ |  |  |  | SM | General | Seq | S | D |  |  |  |
| CPPMem | @ |  | @ |  |  |  | SM | General | Rel | S | D |  |  |  |
| ViP | @ |  | @ |  |  |  | SM | General | Seq | S | D | @ |  |  |
| CDSChecker | @ |  | @ |  |  |  | SM | General | Rel | S | D |  |  |  |
| Coveragecriteria |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Wang et al. | @ |  |  | @ |  |  | SM | General | Seq | U | D |  | @ | @ |
| Hong et al. | @ |  |  | @ |  | @ | SM | General | Seq | S | D |  | @ | @ |
| Bita | @ |  |  | @ |  | @ | MP | Actors | Seq | S | D |  | @ | @ |
| Heuristics |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Rapos | @ |  |  | @ | @ | @ | SM | General | Seq | S | D |  | @ | @ |
| PCT | @ |  |  | @ |  | @ | SM | General | Seq | S | D |  | @ | @ |
| Gambit | @ |  |  | @ | @ | @ | SM | General | Seq | S | D |  | @ | @ |

延伸到新的范例von praun和gross，oopsla 2001[45]。von Praun和Gross利用面向对象程序的封装特性来减少工具的时间和空间开销，以优化锁集分析。该方法利用了限制属性，该属性表示只能由一个执行流访问的对象和对象字段，并在检测期间排除这些对象和对象字段。该技术将静态分析与动态锁集分析结合起来检测内存访问。它在执行的交错中搜索数据竞争，而不预测错误，因此，它不是属性完整的。

雅阁，PPOPP 2011[46]。Accord在基于数组的并发程序中以forkjoin并行性为特征，比如用OpenMP、CILK和TBB编写的程序，瞄准数据竞争。Accord通过依赖一个表示为源代码注释的输入模型来加强锁集分析，该模型指定了预期的并发协调策略。该模型指定了每个执行流读取和写入的内存位置集，区分了仅在获取锁时才应发生的访问和可能在没有互斥的情况下发生的访问。Accord是少数利用开发人员的注释来提高结果准确性的方法之一。Accord使用约束解算器自动验证并发协调策略是否无数据争用。它还自动生成断言，以检查在测试期间实现是否符合指定的并发协调策略。由于约束求解可能无法在所有情况下生成解决方案，因此Accord既不是完整的属性，也不是健全的属性，因为规范只针对基于锁的同步机制。

5.1.2低水位发生在

发生在70年代末Leslie Lamport引入分析之前[47]，并在早期的几个技术中被采用以检测并行程序中的数据竞争[48]，[49]。

在分析之前发生比锁集分析更精确，因为它可以处理基于锁的同步之外的任何类型的同步机制，从而以额外的计算成本避免锁集分析中不可避免的许多误报。在过去的十五年中，在分析被广泛研究之前发生在测试数据竞赛的背景下，重点是（i）提高性能，（i i）减少误报，（i i i）提高完整性，（i v）根据特定的编程范式或语言调整分析，以及（v）将分析扩展到宽松（非顺序）的M。埃默里模型。

不同的方法通过在运行时维护额外的信息（fasttrack[50]），将昂贵的监控活动减少到样本的子集（literace[51]、pacer[52]、sos[53]和carisma[54]），利用多核系统的并行性（wester[55]），以及实现happe来提高分析前的性能。硬件分析前的NS（REENACT[56]）。

一些方法通过（i）对有害和良性的数据种族进行分类来减少假阳性的数量（Narayanasamy等人[57]、Frost[58]和Portend[59]），（ii）开发特定于程序的同步机制（Tian等人[60]）或（iii）考 （RDIT[61]）。Smaragdakis等人drfinder通过考虑锁定代码区域的可选交错来提高分析前发生事件的完整性[29]，[62]，而rvpredict通过将happens before分析与控制流信息集成来提高完整性[30]。

一些技术通过将分析范围缩小到特定类型的应用程序，并利用域语义更精确地推断顺序约束来提高分析的精度：WebRacer[63]和EventRacer[64]分别将分析适应于Web和基于事件的应用程序；DroidRacer[65]将分析应用于Android应用程序。对。

CraceFrand（66）和松弛程序[67 ]利用模型检验增强了分析，以在松弛的Java内存模型中捕获顺序关系。

提高性能FastTrack，PLDI 2009[50]。FastTrack在分析之前引入了一个具有恒定时间和空间复杂性的运行时，从而提高了传统矢量时钟方法的线性复杂性。FastTrack的基础是观察到，在数据竞争检测的环境中，不需要向量时钟的完全通用性来描述大部分读写操作。因此，它提出了一种轻量级的“发生在”之前的信息表示，该信息只记录每个数据项上最后一次写入操作的信息。通过这种方式，它可以将矢量时钟比较的成本降低到与原始happensbefore分析相比的一个数量级。FastTrack既不是完整的属性，也不是健全的属性。

Literace，PLDI 2009[51]。Literace是第一种引入采样以减少分析开销的技术。Literace工具仅用于定义为访问频率较低的代码元素的冷区域，基于频繁访问的代码元素（热区域）参与数据竞争的概率较低的假设。Literace利用属性完整性（检测到的数据争用数）来提高效率。该技术不是属性健全的，因为在分析之前发生的情况可能不知道特定于程序的同步机制。

Pacer，PLDI 2010[52]。与Literace相似，Pacer通过采样来确定分析前发生的效率。与依赖启发式方法的Literace不同，Pacer根据采样率估计在代码区域内查找数据竞争的可能性，从而以成本为代价交换精度。Pacer通过交替执行经典FastTrack算法的采样周期和减少记录信息量并简化矢量时钟管理的非采样周期来扩展FastTrack。非采样周期减少了分析前精确发生的开销。Pacer不是属性声音，因为在分析之前发生的事件不会捕获特定于程序的同步机制。由于采样方法不分析整个执行过程，因此它不是属性完整的。

SOS，OOPLA 2011[53]。SOS通过减少分析前动态发生的开销，提高了快速跟踪和起搏器，同时保持了原始方法的精度。SoS引入了静态对象的概念，这些对象仅在初始化期间写入后才被读取，并且通过从分析中排除静态对象来优化分析前的情况，因为它们不能导致数据竞争。SoS将FastTrack的平均开销减少了45%，同时增加了检测到的争用数量，当考虑到50%的运行时开销时，显示的争用比Pacer多5倍。

Carisma，ISSTA 2012[54]。Carisma利用对同一数据结构（如数组或列表）的多个访问之间的相似性，提高了Literace和Pacer的性能和准确性。在处理同一数据结构的多个访问时，Literace和Pacer等技术使用冗余的内存访问采样来浪费时间。Carisma动态推断指定内存位置和高级数据结构之间映射的应用程序上下文，并使用上下文计算跨数据结构的内存位置分布，以更好地平衡采样预算。基于FastTrack，Carisma既不健全也不完整。

Reenact，ISCA 2003[56]。REENACT通过提出一个原始的硬件实现来改进分析前发生的性能。它将一个程序执行分割成epoch，在每个epoch之前将该程序的状态保存在缓存中，并且如果不受数据竞争的影响，则只在该epoch执行结束时持久保存该epoch的状态。否则，REENACT将回滚，恢复以前的状态，并在同一交错状态下重新执行epoch，使用附加的工具来为开发人员收集有用的信息。REENACT在基于向量时钟的分析之前实现，并将分析开销减少到平均5.8%的程序执行时间。REENACT自动修复与一组预先定义的错误模式匹配的数据争用问题，例如丢失的锁解锁同步，并依赖程序注释来区分良性数据争用。因此，它既不是属性声音，也不是属性完整，因为它不是预测性的。

减少误报。Narayanasamy等人，PLDI 2007[57]。Narayanasamy等人的技术通过自动将检测到的种族分类为良性或有害种族来降低假阳性率。对于给定的数据争用，该方法在数据争用中涉及的内存操作之间重放对不同顺序的执行，并且仅当执行导致不同的程序状态时才将争用分类为有害的。这种方法仍然会导致错误的分类，因为它会错误地将两个交叉点之间的数据竞争归类为良性，这两个交叉点导致无法区分的状态，结果只是偶然的[59]。由于动态分析中的不精确性，该方法不具有完整性。它具有良好的属性，因为它检查数据竞争中涉及的内存操作之间的两个不同顺序是否可行。

弗罗斯特，SOSP 2011[58]。Frost通过比较结果和通过执行具有不同交错的同一程序的多个副本获得的程序状态来检测非良性数据竞争。弗罗斯特将一个执行过程分成几个阶段，并在三个副本上运行每个阶段。它执行一个在分析前用dynamic happens检测的复制副本，以检测程序中的同步点，另外两个复制副本在单个线程上使用一个非抢占控制调度程序。Frost将这两个副本安排为尽可能互补的，这意味着只要同步机制允许，它就会以相反的顺序安排这两个副本中的两个语句。Frost通过比较三个副本的输出，推断出存在有害的数据竞争，并确定最有可能出现故障的副本。当在运行时使用时，frost还可以通过回滚并重新执行错误的epoch来从错误的执行中恢复。霜冻有利用价值成本为3，因为它执行每个时代的三个副本。同时，在备用核心上执行副本可以将增加的利用率降低到3%到12%的开销。由于分析前发生的不精确性和分时机制的不精确性，霜冻既不是一种性质，也不是完整的性质。

Portend，ASPLOS 2012[59]。Portend提出了一个精确的数据竞争分类，其中包括基于它们对被测系统的影响的不同类型的有害数据竞争。它解决了数据竞争分类方法的局限性，这些方法依赖于由数据竞争引起的一对交错的输出和状态的标识：意外标识和不依赖于数据竞争的差异。Portend认为只有在所有可能的测试输入下都产生相同的结果和状态时，数据竞争才是良性的，并使用符号执行检查此属性。由于动态分析的不精确性，Portend的属性不完整。它具有良好的属性，因为它检查数据竞争中涉及的两个内存操作之间的两个不同顺序是否都可行。

Tian等人，ISSTA 2008[60]。Tian等人的动态技术提出了一种自动机制来推断程序特定的同步机制，以提高分析前发生的准确性，从而降低误报率。具体来说，Tian等人基于标志的目标同步，测试并设置锁和障碍。它们推断出表明存在这种同步机制的常见模式，并动态地检测这些模式。该方法提高了数据竞争检测的精度，但没有检测到所有可能的同步约束，因此不能保证属性的完整性或完整性。

RDIT，FSE 2015[61]。RDIT通过分析丢失的事件来减少分析前发生的误报数量，这些事件不是由分析前发生捕获的，因为它们属于无法检测的代码部分，例如外部库。失踪事件是造成许多误报的原因。RDIT通过在关系之前进行扩展来解决这个问题，同时考虑到外部函数的调用，即使它们的实现不能被检测到。RDIT假定在某些共享内存位置上运行的两个外部函数可能会将这些位置用于同步目的，并在两个函数调用之间的关系之前添加一个发生的位置，从而减少如果忽略此关系可能生成的误报数。RDIT依赖于RVPredict，在本节的其余部分中讨论，用于基于关系之前的计算结果进行数据竞争检测。因此，它既不能保证属性的完整性，也不能保证属性的完整性。

提高完整性。Smaragdakis等人，Popl 2012[29]。Smaragdakis等人观察到，通过分析单个执行跟踪，在分析之前发生可能推断出不正确的顺序关系，从而错过一些数据竞争。实际上，在分析推断释放和随后获取同一锁之前发生的顺序关系可能在其他执行跟踪中被违反。Smaragdakis等人通过提出一种新的因果先验关系（CP）来缓解这个问题，该关系捕获语句的执行顺序。cp关系通过推断两个受锁保护的块之间的顺序来松弛happensbefore关系，前提是它们包含冲突的语句。Smaragdakis等人检测当两个冲突的内存访问与CP无关时发生的CP争用，并证明CPrace始终对应于与数据争用的可行交错。该技术既不具有完整的属性，也不具有健全的属性，因为它可能由于特定于程序的同步机制而丢失关系。

Drfinder，FSE 2015[62]。Drfinder针对的是检测隐藏数据竞争的问题，即在分析之前发生的数据竞争，以及在RVPredict中实施的因果先验分析和精确分析，由于分析的过度约束性，可能会错过。隐藏的数据争用由对相同共享内存位置的访问组成，这些访问发生在关系之前，只针对所有可能交错的子集。例如，当关系之前发生的事件取决于获取锁的顺序时，可能会发生隐藏的数据争用，锁可以从执行更改为执行。Drfinder的关键直觉是，通过颠倒关系之前发生的一个或多个操作的执行顺序，可以检测到许多隐藏的争用。例如，颠倒两个锁定操作的执行顺序可能会删除一个happensbefore关系并暴露隐藏的数据竞争。drfinder引入了一个新的may-trigger关系，指定函数是否可以通过函数调用链直接或间接触发锁获取。Drfinder是一种预测技术。它计算执行跟踪上的may-trigger关系，查找可能公开数据竞争的可选交错，并执行选定的交错以检查其可行性。Drfinder既不是完整的属性，也不是健全的属性。

RVPredict，PLDI 2014[30]。rvpredict定义了一个顺序关系来检测数据种族，从而提高了经典分析之前发生的准确性，以及Smaragdakis等人的因果关系分析。对提议关系的关键洞察在于考虑控制流信息。以前的动态数据竞争预测方法通过依赖可能错过一些有效交叉的保守可行性定义，排列执行跟踪，以识别可能遭受数据竞争的可行跟踪。特别是，它们假定读写一致性，这意味着有效排列中的每个读取都返回与原始跟踪中相同的值。rvpredict利用一个更精确的可行排列定义：它将顺序关系编码为一组约束，并调用约束解算器检测种族。由于附加的信息，RVPredict检测到的并发性故障最多比基于分析之前发生或因果关系之前发生的方法多两个数量级。RVPredict不是属性完整的，因为它依赖于约束求解，并且不是属性健全的，因为它可能会错过特定于程序的同步机制，就像所有基于经典的技术都是在分析之前发生的一样。

根据特定范例进行裁剪。Webracer，PLDI 2012[63]。WebRacer通过利用Web平台的语义，特别是有关不同浏览器的规范和实现的信息（如加载、解析和执行之间的顺序约束），增强了分析前发生的情况。Webracer主要关注（i）变量争用，它表示对共享内存位置（如javascript变量）的并发访问导致的数据争用，（i i）HTML争用，它发生在访问表示HTML元素的DOM节点时，在它们创建之前和之后都可能发生；（i i i）函数争用，它发生在函数调用时。可能发生在和之前在对函数进行分析之后，（iv）事件调度争用，当事件在添加相应的事件处理程序之前和之后都可能触发时发生。WebRacer既不是属性健全的，因为在分析之前发生的操作可能会错过特定于程序的同步机制，也不会完成属性的操作，因为它不会检查所分析的执行跟踪的可选交错。WebRacer自动生成与网站交互的事件集，从而生成具体的测试用例。

Eventracer，OOPSLA 2013[64]。EventRacer对WebRacer进行了扩展，为基于事件的程序制定了“发生前”关系，并改进了数据争用的分类，以减少误报。EventRacer将分析扩展到开发人员经常用于扩展Web平台中可用的少量同步原语的即席同步机制，从而消除了数据争用检测器（如WebRacer Report）的许多良性数据争用。eventracer通过引入种族覆盖的概念来修剪良性种族：如果A上的种族被用作同步元素来消除B上的种族，那么A上的种族覆盖B上的种族。对未发现的种族进行廉价的检查可以快速识别同步变量上的种族，并消除这些种族覆盖的假阳性。具有未覆盖争用的数据项集平均比具有争用的所有数据项集小14倍。EventRacer实现了一种有效的向量时钟算法，在基于链分解的分析之前执行[68]。

Droidracer，PLDI 2014[65]。DroidRacer动态地利用Android编程模型的并发语义，在关系发生之前获得精确的结果，从而减少Android应用程序的误报数量。DroidRacer考虑了从多线程执行和异步任务派生的非确定性，这是一种在Android中实现的并发机制，用于防止在用户界面线程上运行复杂的计算。虽然DroidRacer是迄今为止专门为Android应用程序设计的唯一技术，但基于异步回调的Android并发模型与WebRacer和EventRacer针对的Web平台模型类似。与EventRacer相比，DroidRacer不考虑特殊的同步机制，因此它不是属性健全的。它也不是完整的属性，因为它所依赖的分析前发生的限制。

扩展到宽松的记忆模型。Java CraceFrand，ASE 2009 [ 66 ]。与大多数数据竞争检测技术不同，Java ReaveFrand（JRF）不假设顺序一致的内存模型，并在分析之前引入新的JAFSENS，以捕获宽松Java内存模型中的排序关系。JRF依赖于Java探路者模型检查器来生成可能导致数据竞争的交织，并探讨由增加识别数据竞争的概率的模式驱动的交织空间。JRF是一个完整的属性，因为它依赖于模型检查，并且利用了在分析之前发生的、不会产生错误顺序约束的精确变量。它不是属性声音，因为它可能会错过特定于程序的同步机制。

放松剂，ISSTA 2011[67]。与Java ReaveFrand相一致，PraceR定义了一种动态分析方法来检测宽松内存模型中的数据竞争。该分析通过计算在周期之前发生的一组潜在的数据竞争来检测顺序一致的执行跟踪中的潜在的数据竞争，这些潜在的数据竞争表示可能的违反顺序一致性，使用检测到的竞争来预测宽松内存模型上的可选交错，并利用有偏差的随机调度程序强制发生这种交错。Releaser不是属性完整的，因为它使用的随机调度程序可能会错过一些数据竞争中的交错，并且不是属性健全的，因为它可能会错过特定于程序的同步机制。

5.1.3低电平混合

混合技术将锁具和分析前发生结合起来，从分析前发生的准确性和锁具分析的效率中获益。在Choi等人的开创性工作之后。[69]，混合技术限制了在分析代码片段之前发生的代价范围，静态或动态锁集分析可以有效地隔离可能受数据竞争影响的代码片段。

这些方法不同于它们的关注点，这些关注点可以是：（i）通过特定的执行框架提高性能，（i i）减少误报的数量，（i i i）针对特定的同步机制，（i v）覆盖宽松的内存模型，以及（v）完成与测试用例确定的交叉。

与Choi等人、Wester等人的开创性工作一致。RaceMob通过对多核硬件[55]的分析和众包分别在多个执行中分发数据争用检测[70]来提高性能。RaceFuzzer[71]通过随机生成的执行来减少假阳性的数量，这些执行公开数据竞争并观察其效果。racetrack[72]针对基于锁的和fork-join的同步原语，而goldilocks[73]针对软件事务的同步机制。多种族[74]涵盖了放松的记忆模型。simrt[75]用测试用例优先级补充交错，racageddon[76]和narada[77]用测试用例补充交错。

精液工作。Choi等人，PLDI 2002[69]。Choi等人通过结合静态和动态优化技术减少了数据竞争检测的开销。它们通过静态锁集分析确定一组可以导致数据争用的语句，并通过仅跟踪所选语句的运行时内存访问来验证精简语句是否导致数据争用。它们在语句之间引入了弱于依赖关系，这些语句标识了其检测将是冗余的语句，它们缓存内存访问以进一步优化分析。这些优化的结合将运行时开销降低到13%到42%，这是因为为了实现完整性而牺牲了性能。弱关系保证报告在内存位置上的数据竞争中涉及的至少一个（不是每个）内存访问操作。因此，该技术既不具有完整性，也不具有完整性。这是不健全的，因为它可能会错过特别的同步机制。O'Callahan和Choi后来通过动态锁集分析来扩展这种方法[42]。

提高性能。Wester等人，Asplos 2013[55]。韦斯特等人。提出一种并行基础设施，利用多个核心加速锁集，并在分析之前发生。该技术将执行跟踪分为多个阶段，这些阶段在管道中的不同核心上执行。初步的廉价分析确定了其初始条件取决于其他时期的结果以确定初始化所有epoch所需的信息。该技术利用多个核心并行执行不同的时期，并保证一个时期完全在一个核心上运行，从而显著优化分析，因为存储单个时期信息的数据结构的同步性有限。该技术继承了其用于数据竞争检测的工具的局限性，因此既不完整也不健全。

Racemob，Sosp 2013[70]。racemob通过众包优化动态分析。racemob通过锁集分析静态地识别候选数据竞赛，并将这些候选数据竞赛的动态验证分发给许多人。每个人只监视与候选人种子集相关的内存访问，因此只需要为每个人检测一小部分内存访问，从而可以降低单个执行的开销。RaceMob是一款性能优良的产品。它不是完整的属性，因为它不是可预测的。

减少误报。雷射引信，PLDI 2008[71]。Racefuzzer扩展了混合锁集，并在Choi等人提出的分析之前发生。预测替代执行中的数据竞争[42]。Racefuzzer使用不精确但有效的锁集组合动态计算订单信息，并在分析之前进行，以降低计算成本。然后，它使用概率算法计算执行跟踪的可选交错，该算法在尝试执行对共享数据项的无保护访问时，用随机休眠暂停执行流。RaceFuzzer执行由数据竞争决定的不同交错，并使用程序崩溃或断言来区分错误和良性竞争。Racefuzzer不是完整的属性，因为它选择了具有概率机制的交错，也不是完整的属性，因为混合分析可能会错过某些同步约束。然而，最终执行阶段保证了与Oracle相关的可靠性。

针对特定的同步机制。赛马场，Sosp 2005[72]。racetrack检测在Microsoft公共语言运行库的虚拟机上执行的共享内存程序中的数据争用。racetrack通过一种支持基于锁和分叉连接同步的混合算法提高了分析的准确性，并监控内存访问，直至单个对象字段和数组元素的粒度。racetrack通过实现一种自适应方法来减少运行时开销，该方法动态地调整检测的粒度和为每个单元维护的内存访问历史量。racetrack分析系统执行以检测可能的数据争用，而不预测新的争用，既不完整也不健全。事实上，RaceTrack可能会因为它采用的调整分析粒度的自适应机制而错过一些数据争用，并且可能会由于用户定义的锁定机制未被“发生前”关系捕获而生成错误警告。

金凤花，PLDI 2007[73]。goldilocks是第一种考虑软件事务以及其他同步机制的数据竞争检测技术。与Choi等人的方法类似，goldilocks利用一个声音锁集静态分析来避免检测保证无种族限制的内存访问。goldilocks使用考虑内存事务语义的“在分析之前发生”动态检测数据争用。这样，金发姑娘结合锁具和happensbefore分析，提高精度和效率。该技术不能预测交替交织，因此不完整。它也不健全，因为它可能会错过特殊的同步机制。

包括放松记忆模型。多种族，PPOPP 2003[74]。通过考虑松弛的C++内存模型，多种族检测变量和对象粒度水平上的数据竞争。multirrace结合了lockset和happensbefore分析，并考虑了基于锁和屏障的同步机制。多种族旨在检测生产模式下的数据种族，并引入分析优化，以减少对内存访问的检查次数，从而减少开销。多种族既不是属性完整也不是完整的，因为它不是预测性的。它不是属性声音，因为它可能会错过特定于程序的同步机制。

完成与测试用例的交错。Simrt，ICSE 2014[75]。Simrt针对回归测试。它通过识别受程序更改影响和被修改程序中的多个执行流访问的变量，根据在程序更改后暴露新引入的数据竞争的概率来选择和优先处理回归测试用例。它实现了一个经典的贪婪的测试用例优先级，以早期检测可能的数据竞争，从而减少了分析时间，节省了高达95%的总测试时间。Simrt建立在RaceFuzzer之上，以检测数据竞争，并提供相同的保证。

Racageddon，PPOPP 2014[76]。racageddon引入了竞争导向调度，这是一种生成测试输入以及导致执行目标数据竞争的交织的技术。racageddon结合了由O'callahan和choi[42]引入的混合数据竞争检测技术，以识别一组候选数据竞争，并使用动态符号执行来查找一个输入和一个交错，以公开候选数据竞争。racageddon交替使用一个改进的函数来执行动态符号，该函数在调度中排列一些指令，以增加观察候选数据竞争的可能性。racageddon既不是一个完整的属性，也不是完整的属性，因为它继承了底层分析和探索方法的局限性。

Narada，PLDI 2015[77]。Narada自动生成一个测试套件来检测数据竞争。它通过锁集分析来监视顺序测试套件的执行，以识别对共享数据元素的未保护访问，并推断系统的状态以及可以触发数据竞争的函数调用序列。它使用这些信息来合成可以公开数据竞争的并发测试用例。Narada依靠Racefuzzer[71]来执行和分析生成的测试用例。

5.1.4高水平

从直接共享内存访问分析到复杂数据结构分析的方法很少。Shacham等人Colt[78]和Dimitrov等人[79]依赖系统规范来检测高级别的数据竞争，即涉及复杂数据结构的数据竞争。

科尔特，oopsla，2011[78]。COLT通过分析单个执行流的踪迹和随机生成对手并发执行流来检测Java对象上的非线性化操作序列。它通过修剪交错的空间利用数据结构上操作的交换性和一组指定的线性化点，这些点是假定操作发生的程序位置[80]。colt不是完整的属性，因为它依赖于对手执行流的随机定义，并且只执行所有可能交错的子集。它不是属性声音，因为由于程序的语义，随机生成的操作序列可能不可行。

Dimitrov等人，PLDI 2014[79]。Dimitrov等人目标交换数据竞赛，是在同一对象上的一对操作，这些操作不是按照happensbefore关系排序的，也不是按照表达式ecl逻辑公式中给出的交换性规范进行的，后者以每个方法的参数和返回值为基础。该技术将交换性规范转换为中间可执行表示，以丰富“先发生后关系”并减少需要考虑的交错量。该方法不是属性健全的，因为在分析之前发生的情况可能会错过某些同步约束。它不是属性完整的，因为它只检查给定跟踪中的数据争用，不考虑可选的交错。

5.2基于属性：原子性违规

我们根据所考虑的原子性约束将原子性破坏技术分为以代码为中心、以代码区域为目标和以数据为中心、以数据项为目标。与数据竞态检测技术类似，原子性破坏技术通过动态分析目标系统来研究同步机制施加的顺序关系，利用某种形式的锁集或在分析之前发生的情况，并受到地面分析的限制。

5.2.1以代码为中心

检测以代码为中心的原子性冲突涉及（i）识别打算成为原子的代码区域，以及（i i）检测违反已识别代码区域原子性的交错。

不同的技术通过（i）系统规范或模型，（i i）对原子块的假设或启发式方法，或（i i i）推断原子块的静态或动态分析来识别打算成为原子的代码区域。

一般来说，检测交织是否违反原子性对应于检查交织是否可序列化，也就是说，其执行结果等同于任何串行执行的结果。由于即使在小程序中也存在大量的串行执行，因此检查串行性是不切实际的。以代码为中心的方法通过搜索对不可序列化的充分但不必要的条件进行编码的内存访问模式来减少验证交错原子性的问题，从而在性能上交换完整性。当前的方法利用某种形式的动态分析，通常是在分析之前发生的。

以代码为中心的两阶段检测原子性违规的方法源于雾化器，这是Flanagan和Freund的开创性工作[27]。最新的以代码为中心的方法改进了最初的原子化方法，其技术包括：（i）推断原子区域，（i i）减少误报的数量，（i i i）改进性能，和（iv）增加了测试用例生成的交错选择。

SVD[81]和Avio[36]根据被测系统的结构和动态行为信息推断原子代码区域。Velodrome、Atomfuzzer、Have和Penelope通过考虑静态信息（Have[82]）或分析导致原子性违规的交错结果（Velodrome[35]、Atomfuzzer[83]和Penelope[37]）来减少假阳性数量。Wang和Stoller[84]、Ctrigger[85]、Falcon[86]、Best[87]和Doublechecker[88]的方法在许多方面提高了性能。Wang和Stoller利用面向对象的属性来优化雾化器方法；Ctrigger根据交错发生的概率对它们进行优先级排序，从而缩短了揭示原子性违规的时间；Falcon限制了分析过程中存储的信息量，从而交易性能的准确性；在对等价类进行筛选，每个类只检查一个具有代表性的交错；DoubleChecker将不精确但有效的分析与精确但更昂贵的按需分析结合起来。入侵者[89]通过生成暴露原子性冲突的测试用例来完成交织。

精液工作。雾化器，Popl 2004[27]。雾化器是第一个针对原子性违规的工作。雾化器利用锁集分析，特别是一种简化算法来识别不可序列化的交错，依赖于指定原子代码块的代码注释。该技术不是预测性的，因为它只分析执行的交错。

推断原子区域。SVD，PLDI 2005[81]。SVD通过利用数据和控制依赖性自动推断原子区域的近似值。因为SVD原子区域是最大的一组相互关联的、依赖于控制的语句，它们不会再读取该区域中写入的任何变量。原子区域的启发式推理不依赖于任何同步构造，因此即使在错误地省略锁时，它也会标识原子区域。在违反原子性的情况下，SVD报告有关数据项冲突的详细日志，以简化软件调试和修复。由于用于推断原子区域的启发式方法，SVD既不是属性完整的，也不是属性健全的。

Avio，Asplos 2006[36]。与SVD类似，AVIO自动推断原子代码块，但引用了原子性的宽松定义，以检测涉及写-写冲突的原子性冲突，SVD启发式不考虑这些冲突。AVIO根据从系统执行迭代推断的访问交织不变量对原子性进行建模。AVIO从一组初始候选变量开始，对程序中的所有访问对保持不变，并重复执行系统，以删除不可分割的交错，这些交错不会导致系统错误行为。AVIO监控系统的执行，以检测四种不同的访问模式，这些模式违反了访问交织不变量，因此不可序列化。作者介绍了AVIO的软硬件实现。硬件实现利用硬件缓存一致性协议，实际上消除了分析开销，类似于REENACT数据竞争检测器。

减少误报。维洛德罗姆，PLDI 2008[35]。Velodrome通过依赖于检测（而不是预测）原子性违规的原子代码块的规范。Velodrome在“发生前”图中查找循环模式，该图表示原子性冲突的充分必要条件[90]。Velodrome通过修剪无关的操作来定位最有可能导致原子性冲突的操作。

Atomfuzzer，FSE 2008[83]。Atomfuzzer利用注释来指定哪些代码块是原子的，并将分析限制为使用单个锁的成对执行流，以确保代码区域的原子性。它通过利用分析前发生的事件来捕获执行流之间的顺序关系，为测试用例随机生成交错。它以访问关键内存区域的随机暂停来执行测试用例，以最大化观察原子性冲突的概率。Atomfuzzer既不完整也不完整，因为它的预测依赖于概率算法。这是合理的，因为它的最终输出只包含导致Oracle冲突的执行。

2009年3月[82]。通过考虑控制流，扩展了预测技术：它实现了一种混合静态和动态分析，推测性地近似可能发生在尚未在执行跟踪中观察到的分支中的结果。静态地为系统中的每个方法推断一个静态摘要树，以对该方法的控制流、内存访问和同步原语进行建模。将静态摘要树与动态“分析前发生”相结合，生成一个模型，该模型捕获在执行期间发生的事件以及在不同的未探测分支中可能发生的事件。与其他技术（如velodrome）一样，在模型中寻找不可序列化的循环模式。具有不保证可行性，因而不具有财产健全性。它也不是完整的属性，因为它不考虑所分析的执行跟踪的所有可选交错。

Penelope，FSE 2010[37].S Penelope扩展了happensbefore分析，以考虑违反一组预定义原子性破坏模式的锁获取和释放的替代顺序，并在预测的原子性破坏计划下重新执行目标程序，以用测试Oracle删除误报。Penelope不是完整的属性，因为它只检测所观察到的执行跟踪的所有可选交错的子集。它不是属性健全的，因为它实现的动态分析可能会遗漏一些排序约束。

提高性能。Wang和Stoller，TSE 2006[84]。王和斯托勒提出了第一种预测而不是检测原子性违规的技术。他们提出了两种优化面向对象程序中原子性冲突检测的方法。第一种方法扩展了基于lipton的reducation[18]的雾化器，并引入了一些优化，例如识别只读变量和线程局部变量以在线检测原子性冲突。第二个原因是代码块的交换性。它更精确，但也更昂贵，因此离线工作。由于不考虑特定于程序的同步机制，因此建议的方法是完整的属性而不是健全的属性。

Ctrigger，Asplos 2009[85]。和速度场一样，电子测速仪也是一种预测技术。它以C程序为目标，并引用涉及成对执行流的特定访问模式。Ctrigger执行一个测试用例，并依赖分析前发生的事件来预测揭示兴趣模式的交错。Ctrigger建立在这样一个观察基础上：当对一个执行流中的共享变量的多个访问彼此接近时，来自不同执行流的交错访问的概率很低。因此，ctrigger根据访问共享变量的距离对交错进行排序，目的是对发生概率较低的交错进行优先级排序。由于分析前发生的局限性，电流计不具有良好的性能。它也不是完整的属性，因为它检测到所有可能的交错，这些交错暴露了所考虑的模式，但仍然可能遗漏一些错误的交错。

猎鹰，ICSE 2010[86]。Falcon基于固定大小的滑动窗口的概念来检测导致不可交换内存访问的可疑模式。它在固定大小的窗口中维护每个共享数据项的访问信息，并使用存储在该窗口中的信息检测可疑的内存访问模式。滑动窗口通过用新的访问替换窗口中最旧的访问，将焦点集中在密切相关的访问上。窗口的大小是参数化的，它决定了性能和分析精度之间的权衡。由于这种权衡，猎鹰不保证财产的完整性和完整性。Falcon根据导致并发错误的概率对访问模式进行排序。它多次执行目标系统，记录失败和非失败执行中发生的内存访问集，并根据其频率对内存访问模式进行排序。

最佳，ASE 2011[87]。通过定义和定位新的不可序列化模式，Best提高了预测技术（如Have和Penelope）的准确性和性能。Best监视系统执行，识别同步约束，并利用SMT解算器派生可能导致原子性冲突的可选交错。它通过应用将交错聚合到等价类中的简化来提高可伸缩性，并且每个类只执行一个交错。BEST的一个显著特征是对未插入指令的二进制代码的分析：BEST不需要源代码或目标程序的任何修改版本，而只需要调试信息，例如将指令映射到源代码行。事实上，通过依赖于代码的结构，包括使用非同步变量和使用这两种变量之间的代码行的最大距离，可以最好地推断代码块的原子性。由于在分析和原子性假设的推论中的不精确性，BEST不能保证属性的稳定性和完整性。

Doublechecker，PLDI 2014[88]。双重检查通过结合精确和不精确的分析来降低检测原子性违规的成本。不精确分析跟踪执行流之间所有可能的数据依赖关系，并确定一组可能的原子性冲突，由于分析中的近似值，这些冲突包含许多误报。与velodrome和have类似，doublecker precision analysis在happensbefore图中查找循环模式，并通过考虑与不精确分析确定的原子性冲突有关的执行流，精确跟踪数据依赖性。这样，由于分析范围有限，DoubleChecker可以用很小的开销减少第一步中产生的许多误报。DoubleChecker没有完成属性，因为它检测到但预测原子性冲突，并且不是属性听起来，因为在分析之前发生可能会错过一些同步约束。

完成与测试用例的交错。入侵者，FSE 2015[89]。入侵者自动生成测试用例来检测多线程Java库中的原子性违规。与narada类似，入侵者执行一个顺序测试套件，并对目标系统进行检测，以分析锁的获取和释放以及现场访问。然后，它执行基于锁的分析，根据已知不可序列化的四种内存访问模式推断可能的原子性冲突。根据该分析的结果，入侵者将测试套件中可用的顺序测试用例组合在一起，以生成并发的测试用例，这些测试用例很有可能暴露原子性违规。入侵者关注测试用例的生成，并依赖其他技术来检测原子性违规。因此，在表2中，我们没有指定任何依赖于所采用的检测技术的方面（表中不适用）。

5.2.2以数据为中心

检测以数据为中心的原子性冲突涉及到发现原子集可序列化性的冲突，这是Vaziri等人的一个交错属性。2006年引入，以捕获在并发程序中绑定多个数据项的数据一致性属性[38]。最近的以数据为中心的方法主要集中在（i）分析数据项之间的相关性（muvi[91]），（i i）揭示表征某种原子性破坏形式的数据访问模式（Hammer等人以及assetfuzzer[34]、[92]或（iii）生成回归测试，利用系统版本之间的差异（reconttest[93]）。

分析数据相关性muvi，sosp 2007[91]。muvi考虑涉及多个语义相关变量的原子性冲突。muvi建立在这样一个观察之上：语义相关的共享变量应该以一致的方式被访问和修改。muvi检查相关变量是否在同一个原子区域内被访问，并在存在原子区域内不一致保护的访问时发出原子性破坏的信号。muvi通过静态分析方法识别变量相关性，该方法依赖于以下观察：（i）相关变量通常一起访问，因此它们的访问在源代码中通常是相互接近的；（i i）相关变量在同一函数中访问。muvi考虑两个变量x和y是否相关，如果它们在代码行的最大距离内以最少数量的函数一起访问。muvi依赖于现有的方法来检测和预测涉及相关变量的原子性违规。取决于所采用的具体方法的特性在表2中未作说明。

显示访问模式。Hammer等人，ICSE 2008[34]。锤子等。提出了一种基于自动机的原子集可序列化约束冲突检测算法。它们识别了11种有问题的数据访问模式，这些模式描述了非原子集可序列化执行，并适用于通用共享内存程序。这项技术具有预测性。它静态地标识可由多个线程访问的对象字段，插入已标识的对象以为每个字段动态生成自动机，并依赖自动机检测违反原子集可序列化性的访问模式。该技术依靠伪随机噪声注入产生新的交织。所有生成的交错都是可行的，因为它们从具体执行中观察。由于交错选择过程的概率性，该技术可能会遗漏一些揭示感兴趣模式的交错，因此它不是完整的属性。由于在分析中检测不同执行流中指令之间的顺序关系可能存在不精确性，因此它也不是属性健全的。

Assetfuzzer，ICSE 2010[92]。Assetfuzzer使用一个宽松的“分析前发生”来检测Hammer等人的模式，以及一个概率过程来引导执行朝着分析前发生的相关交错进行。assetfuzzer将RaceFuzzer和Atomfuzzer方法扩展到以数据为中心的冲突，这些方法分别针对数据争用和以代码为中心的原子性冲突。assetfuzzer不是属性完整的，因为它依赖于概率算法来选择交错，并且也不是属性声音，因为在分析可能会错过某些同步约束之前，会发生宽松的情况。

回归测试。重新测试，ICSE 2015[93]。Recontest从一个测试套件中选择一个子集，以优化并发程序的回归测试套件。Recontest通过识别可能受程序更改影响的交错，以及通过选择可能只在新版本中发生并导致原子集可序列化性冲突的交错，减少了在并发系统回归测试环境中需要重新执行的大量交错。Recontest同时执行程序的旧版本和新版本，以收集有关内存访问的信息，并使用这些信息通过更改影响分析检测可能受更改影响的内存访问。它执行识别出的交错，将重点放在新版本而不是旧版本的系统中可能出现的内存访问，以及可以在旧版本和新版本中使用不同并发上下文（保留锁集）执行的内存访问。识别出的交错是可行的，违反了原子集的可序列化性，技术是可靠的。

5.3基于属性：死锁

大多数检测死锁的测试方法都建立在Goodlock[94]和死锁引信[95]的开创性工作之上。goodlock动态地将每个程序执行的锁模式记录为一个锁树，并比较线程对的树，以检测可能导致资源死锁的循环依赖关系。死锁引信阐述了用goodlock识别的潜在死锁，以找到死锁的可行执行。

最新的主要技术建立在goodlock和死锁引信之上，以：（i）优化性能，（i i）用测试用例生成补充交错选择，（i i i）考虑锁以外的同步机制。

magicFuzzer[96]和wolf[97]通过修剪锁树来提高性能，而conlock[98]改进了死锁Fuzzer的随机调度程序以避免人为生成的死锁。Sherlock[99]象征性地执行程序以识别相关的测试输入。OMEN[100]扩展了死锁引信，从顺序的测试用例开始生成并发的测试用例。Armus[101]不考虑锁同步，而是针对由屏障同步原语引起的死锁。

开创性的工作。Goodlock，Spin 2000[94]。goodlock以执行流对之间基于锁的同步为目标。它动态检测正在测试的程序以构建一个锁树，它捕获执行流的锁模式，并比较成对的锁树以检测可能导致资源死锁的循环依赖关系。goodlock不执行已识别的潜在死锁。因此，它不能保证所识别的死锁的可行性，既不是财产健全的，也不是健全的。

死锁引信，PLDI 2009[95]。死锁引信对goodlock进行了扩展，实现了对死锁的预测，从而确定了死锁的实际可行性。死锁引信依靠良好的锁分析，因此只考虑基于锁的同步故障。死锁模糊器使用收集到的信息来指导随机调度程序，从而提高选择暴露死锁的交错的概率。

提高性能。MagicFuzzer，ICSE 2012[96]。magicFuzzer通过引入一种修剪锁图的技术来提高goodlock分析的可伸缩性。magicFuzzer观察到，与锁图中的循环相对应的死锁只包含同时具有传入和传出边缘的节点。因此，它迭代地删除不满足此属性的所有节点及其连接的边。magicFuzzer引入了一种新的算法来分析修剪图，它依赖于一个执行流只能参与死锁循环的单个节点的观察。magicFuzzer根据节点所属的执行流对节点进行分区，并实现一种不探索冗余路径的搜索策略。

沃尔夫，PPOPP 2014[97]。wolf通过在分析前采用happens-before-analysis，对事件之间的顺序关系不可行的候选死锁进行删减，提高了goodlock监控分析特性的准确性。通过这种方式，Wolf减少了待由随机调度程序确认的候选死锁的数量。

康洛克，ICSE 2014[98]。conlock解决了随机调度算法在随机调度程序生成人工死锁时出现的抖动问题。在这种情况下，调度程序挂起执行流，无法进行，但无法确认死锁。这个问题降低了随机调度程序的确认能力，从而影响了方法的有效性。conlock通过在动态分析中捕获的顺序关系之前引入should-occurrent来解决抖动问题。这种关系被随机调度程序用来增加到达的概率，从而确认死锁。

用测试用例补充交错。夏洛克，FSE 2014[99]。Sherlock通过动态符号执行扩展了GoodLock，以选择新的输入和交错。这种改进的分析的目标是检测经过长时间计算后出现的死锁，甚至数百万行代码。Sherlock迭代调用两个Execute和Permute函数。前者通过动态符号执行来计算当前执行的路径条件。后者使用路径条件生成一组执行新计划的新程序输入。这两个函数将反复执行，直到达到导致死锁候选的调度或不可行的调度。

奥曼，Oopsla 2014[100]。OMEN通过利用顺序执行的特性自动生成并发测试用例来显示死锁。Omen采用一种类似于入侵者的方法来生成并发测试用例：它执行一个顺序的测试套件，构建一个捕获锁的锁依赖关系。获取已执行的方法，并生成包含顺序测试用例的方法和参数但在不同的执行流中调用的并发测试用例。例如，顺序执行可以标识一组获取嵌套锁的方法调用，如果从多个执行流执行，则可能导致死锁。

扩展到其他同步机制。Armus，PPOPP 2015[101]。ARMUS针对由屏障同步原语引起的死锁。它引入了一种图形分析技术来检测死锁，对于只采用屏障同步原语，但一般不采用属性完整的程序，这种死锁既是属性健全的，又是属性完整的，因为死锁可以从其他同步机制派生出来。X10和Java的ARMUS的实现以低开销为特征，从而提高了工业规模程序的可扩展性。

5.4基于属性：组合

很少有技术同时针对原子性冲突和数据竞争，并且依赖于在分析之前发生的情况。阿加沃尔等。[102]以静态类型检查作为分析前的补充，关注性能。陈和麦克唐纳[103]通过考虑控制流信息，注重准确性。Kahlon和Wang[104]以及Pecan[105]通过向开发人员提供语言来表示不希望的内存访问模式，将重点放在通用性上。

提高性能。Agarwal等人，ASE 2005[102]。阿加沃尔等。提出一种低开销的测试技术，用于检测数据争用和原子性冲突。它们通过结合静态类型检查和修改的锁集分析来减少测试开销。Agarwal等人的静态类型检查是一种线性时间（尽管不完整）的混合类型发现机制，该机制根据混合数据竞合检测技术（使用锁集分析来隔离未受保护的内存访问，这些访问在分析前进行精确监控），识别大部分受测系统的类型。.agarwal等人的改进锁集分析利用通过类型发现推断的类型信息来有效地检测目标系统中的数据种族和原子性冲突。由于类型发现和动态分析阶段的近似关系，该技术既不是属性健全的，也不是属性完整的。

提高准确性。Chen和MacDonald，ASE 2007[103]Chen和MacDonald的混合分析确定了可能并行发生的语句，并生成执行选定的并发访问对集的交错。Chen和MacDonald的方法根据三种类型的并发访问对、读/写共享数据项、并发获取同一锁、并发在同一监视器上等待/通知选择要执行的相关交错。与Agrawal等人不同，与Have类似，Chen和MacDonald的静态方法考虑了控制流和订单信息，并通过一组测试用例分析了代码的所有部分。由于JPF模型检查器不保证探究所有可能的系统状态，因此该技术是属性健全的，但不是属性完整的。

为模式规范提供语言。Kahlon和Wang，CAV 2010[104]。Kahlon和Wang的方法追溯了并发程序中的因果关系，并将待验证属性引入的约束考虑在内，利用通用因果图（UCG）扩展了happensbefore的概念。例如，在数据争用的情况下，要验证的属性是存在对共享数据项的多个并发访问，而没有同步。该方法通过将从程序同步结构派生的约束与从目标属性派生的约束相结合来修剪大量与目标属性无关的交错。Kahlon和Wang讨论了涉及数据竞争和原子性违规的方法，但是UCG可以扩展以捕获其他自定义属性。该方法不是属性健全的，因为happensbefore分析可能会错过某些同步约束，也不是属性完整的，因为预测阶段依赖于无法解决某些约束的SMT解算器。

Pecan，ISSTA 2011[105]。Pecan引入的关键新颖性是一种简单的语言，将内存访问异常定义为不同执行流与共享数据项交互的模式。模式同时考虑执行的内存操作（读或写）的类型和内存操作所属的原子区域（如果有的话）。模式语言捕获众所周知的内存访问异常，例如数据争用和原子性冲突。pecan实现在分析之前进行，以推断指令之间的顺序关系，为每个已识别的交错创建一个时间表，并检查其可行性。Pecan不保证属性的可靠性，因为在分析之前发生的情况可能会错过某些同步约束，并且预测算法仅在特定的同步模式（嵌套锁）下保证属性的完整性。

5.5基于属性：违反订单

有几种方法可以解决顺序冲突，这些冲突不能简化为前面部分讨论的交错的任何经典属性（数据争用、原子性冲突和死锁）。这些方法寻找来自（i）特定类型的错误，（i i）程序规范或（i i i）编程模型语义的属性冲突。

pretex[106]和2ndstrike[107]处理类型状态错误，即涉及违反数据结构高级语义的错误。conmem[108]、conseq[109]和exceptionnull[40]针对影响程序行为的错误的冲突，例如导致空指针取消引用、悬空指针和无限循环的交错。Maple[110]关注共享变量访问的模式。与其他针对故障类型的方法不同，MAPLE针对可能或可能不会导致并发故障的关键模式。

jPredictor[111]、sinha和wang[112]以及gpredict[113]寻找违反程序规范的并发行为，而defuse[114]针对违反自动生成的数据流不变量的顺序和并发错误。

Simracer[115]针对过程级违规，CAFA针对Android程序[116]，Mutlu等人。目标javascript程序[117]。

违反特定类型的故障。Pretex，ASE 2008[106]。pretex预测并发类型状态错误。type state是与给定类型的对象关联的状态，它定义了一组可以应用于该状态下对象的操作[118]。在不支持op的typestate中对对象obj调用操作op时，会发生typestate错误。关闭文件后读取文件是typestate的简单示例。故障。与许多常见研究的并发性错误不同，例如与低级别内存访问相关的数据争用和原子性冲突，并发类型状态错误与特定目标系统的高级别语义相关。PrTeX工具是用于跟踪事件的目标Java程序，如方法调用和线程创建。在执行测试用例时，pretex（i）计算事件之间发生在前的关系，（i i）确定共享哪些对象，以及（i i i）根据挖掘技术推断每个共享对象的类型状态属性。pretex生成并发执行的有限状态机模型，并检查生成的模型是否存在类型状态属性冲突。pretex不是属性完整的，因为自动推断类型状态规范可能会遗漏某些规范，而且也不是属性健全的，因为它使用的是“发生在前”关系的近似值，该关系可以识别不可行的冲突。

第2次试验，ASPLOS 2011[107]。与pretex一致，

2ndstrike检测涉及文件、指针和锁的并发类型状态错误。该技术动态分析测试用例的执行，以生成一组候选错误，用不同线程中的成对操作表示，如果以交换顺序执行，则可能导致类型状态错误。该技术依赖于发生在关系之前的事件来标识无法重新排序的操作。与pretex不同，2ndstrike使用确定性调度程序强制执行分析期间计算的候选故障。2ndstrike可以扩展到与其他对象一起工作。它只需要（i）一个输入有限状态机，它捕获目标对象的类型状态以及可以在每个状态中应用的操作，以及（i i）一个程序的工具来分析相关的对象方法。该技术具有良好的性能和良好的性能，但由于分析不考虑程序的控制流程，因此无法完成性能分析，因此可能会漏掉一些故障。

Conmem，Asplos 2010[108]。Conmem建立在这样的观察基础上，即基于传统交织特性（如数据种族和原子性侵犯）的测试技术既存在误报，也存在仅根据其影响的严重性来区分故障的不可能。conmem克服了这些局限性，重点关注并发故障的影响，而不是可能导致这种影响的内存访问模式。conmem选择可导致空指针取消引用、读取未初始化数据项和访问无效内存位置的交错，如缓冲区溢出。conmem离线分析执行跟踪，以确定可能导致所考虑类别并发问题的可选交错。conmem为每个选定的交错生成具体的执行，以删除不可行的和误报的交错。由于最初的动态分析，整体方法既不健全也不完整。

康塞克，ASPLOS 2011[109]。conseq通过考虑一组更大的故障类型来扩展conmem，包括断言冲突、无限循环、错误消息过程调用和无效内存访问，以及通过因果关系链触发的间接故障。通过切片和动态分析，改进了conmem方法。它静态地将程序切片到故障中涉及的数据项，以确定对共享数据项的访问，这些访问可以通过控制链和数据依赖关系影响读取值。它动态分析正确的程序执行，以推断可能改变动态控件和数据依赖关系，并导致读取不正确的值。conseq为每个选定的交错生成一个确定的调度，并执行生成的调度来删除不可行的交错和误报。由于静态分析中的近似值，Conseq的属性不完整。

例外情况ull，FSE 2012[40]。ExabutEL检测到可能导致Java程序中共享数据项空指针撤销的交织。exceptionull实现混合锁集，并在接近前发生。它分析执行跟踪以捕获内存访问、操作之间的顺序关系、锁获取和释放。它识别两个不同执行流的事件e1和e2对，这样e1将空值写入共享数据项x，e2读取x。它使用smt解算器检测导致空读取的交错，并在受控环境中执行识别的交错，以检查其可行性。exceptionull同时是属性sound和sound，但不是属性complete，因为SMT解算器可能无法为某些输入约束提供解决方案。

枫树，Oopsla 2012[110]。Maple扩展了检测和预测数据竞争和原子性冲突的技术，使用了六个交错的习语，这些习语表示可疑的内存访问模式，这些模式编码了对顺序约束的一般违反。Maple分析了成对的执行流，假设它们可以揭示大多数并发错误。它捕获与happens before analysis的顺序关系，计算暴露某些已识别习惯用法的可选交错，并使用确定性调度程序执行选定的交错，以查找错误。MAPLE记录执行的交错，但不记录执行过程中观察到的特定值，并且可能会遗漏一些依赖于执行中采用的值的错误。因此，枫树并不是完整的财产。

违反程序规范。JPreditor，ICSE 2008[111]。jPredictor通过从检测数据竞争和原子性冲突的混合技术中获得的两步分析来识别违反程序规范的行为。它将执行跟踪缩小到仅与要用静态分析检查的属性相关的事件，并基于切片因果关系的概念构建涉及所选事件的因果关系图，这是关系之前发生的一个变量，并预测和执行可能导致属性冲突的可选交错。与前几节讨论的检测数据争用和原子性冲突的技术不同，jPredictor检测用户定义属性的更常见冲突。jPredictor是声音和属性声音，因为它执行并检查所选的交错。由于动态分析中的不精确性，它不是完整的属性。

Sinha和Wang，FSE 2010[112]。Sinha和Wang预测可能导致断言冲突或C程序中的数据竞争的交错。与have类似，sinha和wang考虑了控制流关系，并构建了与测试用例执行相对应的并发控制流图（ccfg），捕获了执行流中的内存访问和同步原语，并在执行中为每个违反的断言包含一个错误节点。它们检查每个错误节点，以计算由程序输入上的一组约束组成的路径条件，并指示到达错误节点的语句的顺序。然后，他们用约束解算器解决路径条件，以识别将系统带到错误节点的交错。由于不精确的由于约束解算器可能无法找到导致系统进入错误状态的某些解决方案，因此分析和也不是属性完整的。

Defuse，Oopsla 2010[114]。defuse的主要贡献是从数据流不变量自动生成测试Oracle，以检测顺序和并发故障。defuse考虑（i）本地/远程不变量，该不变量指定数据项上的读取操作仅使用本地或远程执行流的定义，而不使用这两者；本地/远程不变量捕获顺序冲突以及并发程序中的读写原子性冲突；（i i）跟随不变量，该不变量指定从同一执行流对同一数据项的两个连续读取访问始终使用相同的定义；跟随不变量捕获并发程序中的读取原子性冲突；以及（iii）定义集不变量，指定允许给定读取访问查看的写入操作集。defuse通过从大量正确运行中推断不变量来训练分析，并检查在特定交织中是否违反提取的不变量。defuse既不是属性完全，也不是属性声音，因为学习阶段既可以忽略不变量，也可以生成误报。

gpredit，ICSE 2015[113]。与jPredictor类似，gpredict根据语句的顺序验证以正则表达式表示的高级属性。gpredect推断仅依赖线程本地跟踪并忽略全局同步的动态标识在执行跟踪上的事件之间的顺序关系，从而减少动态分析的开销。gpredect通过一个约束解算器来预测可能的并发错误，从而检查是否存在违反并发属性的交错的可行性。gpredit不是属性健全的，因为分析可能会丢失某些顺序关系，并且不是属性完整的，因为约束求解器可能无法为某些约束生成结果。

违反编程模型语义。Simracer，ISSTA 2013[115]。Simracer关注涉及进程间共享资源（如文件和设备）的进程级顺序冲突，并检测涉及用户进程、软件信号、软件处理程序和内核进程的并发故障。该技术具有预测性：它动态地收集有关同步操作的信息，并访问在测试用例执行期间发生的共享资源，在分析之前对收集到的信息进行分析，以识别潜在的顺序冲突，并在受控环境中执行程序以强制执行此类顺序VIO。发生关系。这项技术依靠测试手段来区分良序和错误序。该技术并不是属性完整的，因为它不考虑所有可能的共享资源，这些资源可以由不同的进程访问。它是合理的，因为它执行选定的交错，并将它们与开发人员提供的测试Oracle进行比较。

CAFA，PLDI 2014[116]。CAFA分析事件驱动（Android）程序。在事件驱动的程序中，每个执行流都拥有一个事件队列，并以不确定的顺序处理推送到该队列中的事件。与exceptionnull一致，cafa检测事件驱动程序中发生的特定形式的顺序冲突，并在释放指针后取消引用时导致在自由冲突后使用。CAFA实现在分析执行跟踪之前发生，以确定事件和取消引用指令之间的顺序关系。它是启发性的通过忽略由检查其安全性的分支条件保护的访问，而不是将不能在程序单元外部传播的取消引用操作视为危险的访问，来消除误报。CAFA选择可能导致免费违规后使用的替代交错，并通过不确保输出交错的可行性的离线分析进行报告，因此不具有良好的属性。由于分析中的近似值，CAFA也不是完整的。

Mutlu等人，FSE 2015[117]。穆特鲁等。检测客户端javascript程序中的特定顺序冲突，这些程序执行多个异步请求，由于可能的网络延迟，这些请求的回调可以相互竞争。穆特鲁等。关注XMLHttpRequest，一种用于从服务器请求数据的异步机制。当请求的答案可用时，javascript将调用回调函数。在存在多个请求的情况下，相应回调函数的调用可能以不同的顺序发生，从而可能导致违反顺序。该技术通过关注写在系统持久状态上的回调函数之间的危险竞争（例如在客户端本地cookie上），区分良性和有害的顺序冲突。在这些情况下，执行结束时持久状态的值严格依赖于执行的交错。根据上面的定义，该技术是可预测的，因为它监视目标系统的执行跟踪，以识别可能导致有害种族的交错。这项技术引入了一种在分析JavaScript程序之前发生的特定情况，以检测无法重新排序的事件。该技术不能保证所选交织物的可行性，既不具有良好的性能，也不具有完整的性能。

5.6空间探索

一些测试技术探索了不受特定属性驱动的交错空间：应力测试、详尽探索、覆盖标准和交错空间的启发式探索。

5.6.1应力测试

经典的压力测试方法执行测试套件，在越来越重和极端负载条件下运行目标系统。在测试并发系统的环境中，压力测试方法多次执行相同的测试套件，而不显式地控制调度。最近为数不多的并发系统压力测试技术检查了执行的线性化[119]或更改对性能的影响[120]。

检查线性化。Pike，Eurosys 2011[119]。派克检测语义和潜在的并发错误。语义错误违反了应用程序的语义，例如返回一个意外的值，并且很难找到，因为它们通常不会导致系统故障。潜在故障会破坏系统的内部状态，并且可能比触发时出现得晚得多。派克通过比较并行执行的输出和内部状态与许多可能的线性化的输出和内部状态来检查并行执行的线性化。虽然比较两个执行的输出很简单，但是由于存在嵌套的数据项，比较内部状态很复杂。PIKE基于一个表示为状态摘要函数的规范来比较内部状态，该函数捕获系统内部状态的语义，并允许进行有效的逻辑比较。检查更改对性能的影响。速射枪，ISSTA 2014[120]。在我们的调查中，速度枪是唯一以性能为目标的方法，尤其是性能回归测试。SpultFun针对Java程序，并自动生成一组测试用例，该测试用例由N个不同的执行流程上并发执行的顺序前缀和N后缀组成。speedgun在被测类的旧版本和新版本上执行多次测试用例，并报告两个版本之间的相关性能差异。speedgun多次执行每个测试，以观察各种交错，并减轻外部因素的影响，如垃圾收集和及时编译。它不控制交错以避免对性能的影响。speedgun优化了测试用例的大小，以权衡精度和执行时间。

5.6.2全面勘探

详尽的探索方法通过使用符号执行[124]或模型检查[125]、[126]、[127]、[128]、[129]来分析给定测试输入的所有可能的交错，无论是在一个具有合适测试套件[121]、[122]、[123]的可管理空间上，还是在整个空间上。不同的方法在消息传递MPI程序（124）、基于演员的程序[125 ]、C和C++松弛内存模型[127 ]、[129 ]、不变量违规[128 ]和死锁[126 ]的上下文中研究穷举探索。

探索减少的交错空间。Ballerina，ICSE 2012[121]。Ballerina为并发的面向对象程序生成并执行测试用例。作为大多数方法，Ballerina建立在这样一个假设之上，即使用两个并发执行流可以发现大多数错误[130]。Ballerina测试用例由两个执行流组成，它们将随机生成的序列前缀与并发后缀配对。Ballerina开发了执行线程交错的不同策略，包括穷尽状态搜索、抢占边界、无状态搜索和并行测试执行，并介绍了一种基于其特征的集群线性化违规技术，以简化故障分类和错误检测。积极的结果。Pradel等人[122]扩展Ballerina的Oracle，如果且仅当它们导致系统崩溃或死锁时，才会将非线性执行视为错误。这样，它们只识别导致可见故障的非线性行为。Pradel和Gross[123]进一步扩展了Ballerina，并使用一种技术来预测并发面向对象程序中的可替换性冲突。可替换性是正确设计的面向对象程序所需要的属性。它确保子类的实例始终可以替换超类的实例，而不会对超类的客户端造成行为差异。pradel和gross的方法在一个超类和相应的子类上执行自动生成的测试用例，并将该超类的行为用作子类的Oracle。普拉德尔和格罗斯的预言是行为差异、例外和崩溃。Ballerina通过执行每个选定的交错来识别非线性行为并确保属性的可靠性，但由于生成阶段的概率性质和对两个执行流的限制，因此不完整。

探索与符号执行交织的空间。Sen和Agha，FASE 2006[124]。Sen和Agha的方法自动测试依赖消息传递范式的分布式系统，尤其是MPI程序。该技术生成一组测试用例，执行程序的所有可到达语句，并执行生成的测试用例的所有可行的交错。该技术动态地执行初始测试用例，为所有执行的分支点生成路径条件。它探索了执行的不同交错，这些交错是用只捕获可行交错的“发生在分析之前”来标识的。它通过解决违反分支路径条件的约束来生成新的测试用例，这些测试用例到达尚未执行的语句。该技术使用系统崩溃和死锁作为oracle，并返回暴露此类问题的测试用例和交错。

通过模型检查探索交错空间。巴塞特，ASE 2009[125]。BasSET通过扩展Java探路者模型检查器〔131〕来支持基于消息的程序生成高效的测试用例，以支持消息调度。巴塞特利用动态部分阶约简对勘探状态空间进行了删减，并在分析前发生。动态部分序约简通过停止对与已探索状态相同的状态的探索来阻止等效交织的执行，而在分析之前发生则避免产生不可行的交织。巴塞特还引入了一种优化方法，用存根代替外部库，以加快空间探索的速度，该方法基于这样一个假设：参与者之间的消息交换只受外部库行为的轻微影响。

将死，FSE 2010[126]。checkmate检测死锁，而不依赖于任何特定的同步范式，因此它不同于传统的死锁检测技术，后者针对基于锁的同步派生的死锁。checkmate监视目标系统的执行并构建跟踪程序，这是一个简化的执行模型，包括相关的同步方面。然后，它从执行的程序跟踪生成所有可能的交错。与basset类似，checkmate依赖于模型检查来标识可能导致死锁的交错。checkmate通过构建一个简化的模型来限制分析成本，从而消除与检测死锁无关的计算。模型中引入的近似值使得该方法不可靠且不完整。

CPPMEM，Popl 2011[127]。CPPMEM详尽地研究了测试用例的所有可能的交错，以识别数据竞争和其他交错模式，这些模式可能会在存在宽松的内存模型时导致错误。CPPMEM利用C++严格的语义，用于C++ 0x并发模型，该模型考虑了保证顺序一致性的操作和接受松弛一致性保证的低级别指令，以便能够实现更高级别的代码优化和系统性能。该技术旨在分析可能表现出非顺序行为的小代码片段。CPPMEM精确地定义了顺序关系，但忽略了控制和数据流约束的作用，从而能够识别不可行的交叉。CPMEM不保证可靠性或完整性。依赖nipkin模型检查器的CPPMEM的另一种实现在保持相同目标和保证的同时提供更好的性能[132]。

VIP，FSE 2012[128]。VIP使用VISISOFT模型检查器〔25〕探讨C和C++程序的交错空间识别过去时间线性时间逻辑（PLTL）中表示的用户定义属性的冲突。VIP不保证所选插页的可行性，因此是不健全的。

cdschecker，oopsla 2013[129]。CDSChecker以C++低级指令实现的松弛内存模型为目标，扩展了CPPPMEM，大大提高了其性能。CDSChecker将C++语义所提供的内存一致性保证编码为一组约束，并将这些约束与定义为顺序一致执行的FaseSnFor约束相结合，以修剪冗余和不可行的执行。与CPPMEM类似，CDSCHECKER精确地定义了顺序关系，但忽略了控制和数据流约束的贡献，因此它可以识别不可行的交叉，并且不能保证完整性或可靠性。

5.6.3覆盖标准

覆盖标准通过利用共享内存位置中的数据流关系来确定要探索的程序空间子集（Wang等人[133]），同步代码块的顺序（Hong等人[134]）或actor程序中的消息顺序（bita[135]）。

Wang等人，ICSE 2011[133]。王等。介绍hapset的概念，它描述了一个测试用例的相关交错集。它们将操作op的hapset定义为一组操作op0，在op0和op之间没有任何中间操作的情况下，可以访问至少一个交错操作中与op相同的内存位置。该技术迭代计算一组测试用例的hapset，并探索不同的交错操作，这些交错操作不是执行的hapset。在每一次迭代中，该技术都会用运行时已经研究过并且不会产生任何系统崩溃的所有hapset来增加最初的hapset集，直到不再生成hapset为止。hapset能够很好地捕获常见的并发错误，如数据争用和原子性冲突，因此通过覆盖hapset，该技术可以发现许多并发错误。由于许多交错共享相同的hapset，因此集中于它们在很大程度上减少了要执行的交错量。

Hong等人，ISSTA 2012[134]。Hong等人的覆盖标准侧重于同步块，并要求以不同的顺序执行所有同步对。它们还考虑了属于同一线程的代码块，从而扩展了以前对同步对覆盖率[136]的工作。该方法动态地构建一个线程模型，该模型捕获线程之间的内存访问和同步操作，通过分析该模型迭代地标识尚未覆盖的同步对，并在受控环境中执行尚未执行的同步对。该方法既保证了可行性，又保证了可靠性，但不保证完整性。

Bita，ASE 2013[135]。Bita以基于参与者的程序为目标，并在参与者接收的成对事件上引入三个覆盖标准：“连续接收对、接收对和行为改变对和接收标准”，这要求覆盖不同的接收事件序列。Bita通过选择可根据所选标准提高覆盖率的交错来逐步构建测试套件，并在受控环境中执行它们，同时检查系统崩溃，从而确保可行性和可靠性，但不保证完整性。5.6.4启发式

启发式方法将要探索的交错空间绑定起来，并对它们的执行进行优先级排序：rapos[137]利用部分顺序约简来识别等价的交错，pct[138]限制调度约束的数量，gambit[139]根据它们对执行的交错的多样性对它们进行优先级排序。

Rapos，ASE 2007[137]。Rapos通过利用部分序约简来减少探测空间，这是一种模型检验者用来识别等效状态的优化技术。RAPOS以Java程序为目标，并在CalFuffER之上实现。

PCT，ASPLOS 2010[138]。概率并发测试（PCT）在C和C++程序中检测有纪律的随机测试中的一般并发错误。PCT保证在产生最多n个线程并执行最多k条指令的程序中查找深度d的故障，概率至少为1=n k d 1，其中深度d被定义为需要强制执行以揭示故障的最小调度约束数。

甘比特，PPOPP 2010[139]。gambit以交织空间的压缩表示方式动态记录执行交织之间的差异，并使用该模型根据两个优先级函数对交织探索进行优先级排序：具有进度保证的随机搜索，以及支持在交织之前公开新事件的功能。矿石关系。gambit使用chess模型检查器控制执行，并使用断言冲突和死锁作为Oracle。所选交错的执行既保证了可行性，又保证了可靠性。

6讨论

第5节中对研究状况的详细分析突出了一些相关趋势，我们在本节中根据第4节中介绍的分类标准进行总结，如图4所示。

6.1输入

超过95%的分析方法集中在为给定的测试套件选择执行交错的问题上。并发软件系统的测试用例生成问题虽然很重要，但尚未得到广泛的研究。以生成测试用例为目标的少数技术，要么（i）从一组确定要执行的操作的顺序测试用例中构建并发测试用例，从而将生成并发测试用例的问题减少到在多个执行流上分配操作的问题（并行化方法），要么（i i）gener随机地测试用例，并依赖于大多数并发性错误可以通过少量的执行流和操作触发的假设，因此通常会生成小的测试用例，并简化选择交错（随机方法）的问题。此外，到目前为止，生成测试用例和选择交错之间的交互作用还只是很少的研究。

大约10%的分析方法依赖于系统模型来获取关于系统行为的语义信息。这些模型对系统所需的属性进行编码，并用于指导违反这些属性的交错技术。6.2交错的选择

80%以上的分析方法是基于属性的，其中约60%是可预测的，也就是说，它们分析执行跟踪，以确定可能暴露对相关属性的违反的可选交错。随着时间的推移，预测技术越来越流行，从Sen等人关于预测跟踪分析的开创性论文开始。〔140〕。虽然一些检测数据竞争和原子性冲突的技术不是可预测的，但是几乎所有针对死锁或顺序冲突的技术都是可预测的。

有几种成熟的监控方法可以在运行时检测和避免死锁，5因此测试技术主要用于预测在现场出现之前的交替交错死锁，如表3所示。如表5所示，顺序冲突技术主要针对预期的程序不变量，并预测违反这些不变量的可选交错。如表6所示，勘探技术在研究环境中正逐渐失去普及，并主要由详尽的勘探技术所主导，有时与测试用例的自动生成相结合，以限制执行流的数量和为可行性而要探索的并发操作的数量。压力测试技术由于无法有效地探索交错空间，因此很少受到关注。最后，目前仅提出了几种基于启发式或覆盖准则的方法。

6.3交织物的性质

大多数基于属性的技术处理数据争用和原子性冲突，重点是低级数据争用（表1、2和4）。涉及高级数据结构和编程抽象的模式正日益流行。这是原子集可序列化的情况，由Vaziri等人在开创性论文中介绍。[38]并在Shacham等人的Colt中发表。Dimitrov等人（表1中的行高级别）。

死锁检测在测试中受到的关注较少，因为存在有效和高效的技术来避免、检测和恢复运行时的死锁。

如表5所示，检测顺序冲突的测试技术正变得越来越流行，可能是因为它们的目标属性更好地编码了程序的语义。此外，最近的研究表明，违反顺序大约占所有并发错误的三分之一[130]。此区域中的技术以一般属性（如空指针取消引用）或特定于域的属性为目标。

6.4输出和Oracle

超过65%的选择交错的分析方法不考虑执行已识别序列和将结果与Oracle进行比较的问题。限制识别可疑交叉的努力不执行它们可以降低成本，但会导致许多误报。

如表3所示，大多数检测死锁的方法执行所选的交错，以证明预测的死锁确实是可行的。类似地，许多最新的检测违反命令的技术利用显式或隐式的oracle来区分失败的执行和误报。总的来说，我们观察到降低假阳性率的兴趣越来越大，这是许多方法的主要限制。

6.5保证

超过80%的分析方法保证了所确定的交错的可行性。所有基于属性的检测方法都是可行的，因为它们只关注单个执行跟踪。基于属性的预测技术通过执行选定的交错来保证可行性；这是几乎所有死锁检测技术的情况，如表3所示。如表6所示，绝大多数勘探方法通过执行勘探期间观察到的交错来保证可行性。

属性的完整性和完整性只适用于基于属性的技术。由于以下一个或多个考虑因素，几乎没有一种技术是属性完整的：（i）一些技术只执行检测（而不是预测），因此可能会错过可选交错中的相关属性；（i i）一些技术依赖于过度约束的分析，例如分析前的某些形式；（i i i）一些技术niques依赖于有界模型检查或约束求解来生成可选的交错，这些交错可能无法生成一些有效的交错，（iv）一些技术利用概率方法来生成或执行可选的交错，（v）一些技术的范围有限，例如一些技术关注的是最多涉及两个执行流。

不执行所选交错的技术不是属性声音。事实上，由于所采用的动态分析精度有限，所选的交错可能不可行，这可能会遗漏通过特定于程序的同步机制定义的某些顺序约束。

完整性和完整性适用于将执行测试用例的结果与Oracle进行比较的技术。由于大多数技术都不是属性完整的，所以它们也不完整。几乎所有将测试用例执行结果与Oracle进行比较的技术都是可靠的。由于模型中的近似关系，只有将Oracle实现为对抽象系统表示的模型检查的技术可能无法保证其可靠性。

总的来说，我们观察到在提高故障预测率和降低假阳性率方面越来越有兴趣，（i）提高动态分析技术的精度，以及（i i）针对特定的故障模式或编程范例。

5.这些监测方法超出了我们调查的范围，我们调查的重点是测试技术。我们将在第7节中详细讨论它们。

6.6目标系统

几乎所有最先进的并发系统测试技术都涉及共享内存系统。只有很少的技术（我们在本次调查中讨论其中的五种）处理消息传递系统，这是运行时监视和基于模型的验证研究的主要焦点。监控技术在运行时观察目标系统的行为，以分析所观察到的故障的根本原因或预测未来的故障以防止其发生。基于模型的技术验证了抽象模型上消息传递系统的正确性。我们将在第7节中讨论这两类方法。

大多数最先进的技术都以通用并发系统为目标。很少有例外集中在特定的编程上范例：（i）依赖于采用基于锁的同步机制的锁集分析目标系统的数据争用检测技术（表1中的行低级别锁集）；（i i）大多数死锁检测技术针对源自基于锁的同步故障的死锁（表3）；（i i i）高级别数据争用技术在抽象对象的层次，特别是面向对象的系统（表1中的高层次行）；（iv）一些技术通过依赖特定编程范式中定义的操作语义来提高效率。例如，一些技术专注于基于事件的系统，例如Web平台、Android平台或JavaScript，在这些系统中，框架强制执行操作之间的某些顺序关系。

几乎所有最近的方法都假定一个顺序一致性模型；只有很少的方法（我们在本次调查中讨论了五种方法）考虑了在特定编程语言（如Java和C++）中实现的内存模型。放宽序贯一致性模型的假设仍然是未来研究的一个开放领域。

6.7测试技术

测试并发系统的大多数技术都依赖于对系统的某种动态分析，以获取有关多个执行流中操作之间顺序关系的一些信息。大多数采用的技术发生在分析之前。只有少数方法（我们在本次调查中讨论其中九种）提出了某种形式的静态和动态混合分析，以提高结果的准确性。例如，在原子集可序列化的情况下，一些技术使用静态分析来推断系统的相关属性，例如原子区域或相关变量。

大多数技术关注并发性故障对系统功能的影响，这在很大程度上打开了违反非功能属性（如时间属性或安全属性）的问题。大多数技术应用于系统粒度级别。只有少数技术关注于单元粒度级别，通常是面向对象系统中的对象。大多数技术都集中在验证测试上。只有少数技术（Recontest[93]、Simrt[75]和SpeedGun[55]涉及回归测试。

所有技术都是指集中的测试体系结构，它反映了对消息传递范式的有限关注，主要用于开发分布式应用程序。可能部署在大规模环境中的分布式消息传递系统仍然是一个很大程度上开放的研究领域，涉及高效和潜在分布式测试架构的设计。

6.8有效性威胁

对我们调查有效性的主要威胁来自我们选择回顾文献的策略，如第3节所讨论的。这些威胁包括我们用来选择论文的标准的有效性，遗漏某些相关工作的可能性，以及我们用来界定调查范围的过滤器的一致性。在本节中，我们承认可能限制我们结论有效性的威胁，并简要讨论我们为减轻相关风险而采取的对策。

选择标准。我们根据关键字搜索相关存储库来确定我们的选择。测试技术可能在不同的社区以不同的术语被提及，并且可能已经在不同的场所发布。初始搜索的关键字和存储库的选择对于我们调查的成功至关重要。我们补充了搜索最流行的存储库（ieee explore、acm digital library、springer online library和elsevier online library）通过web进行常规搜索，采用多变量关键字集，包括“testing+concurrent”、“testing+multiphread”、“testing+parallel”、“testing+distributed”。为了减轻任何术语和存储库选择的内在局限性，我们在搜索中还包括引用或引用我们选择的论文的出版物。如图3所示，我们的选择标准使我们能够识别在不同研究群体中呈现的许多结果。

错过工作。我们基于文献库、关键词和论文之间的相互关系进行的文献检索可能遗漏了一些未被引用的论文，也没有引用任何选定的论文。我们通过分析过去15年来发表的会议记录和期刊问题来降低这种风险。

边界过滤器。界定调查范围总是有随意排除某些相关工作的风险。我们通过精确界定调查范围来降低这一风险。我们在第7节中列出了密切相关但不受约束的工作，其中我们介绍了本次调查排除的主要主题，即并发故障的实证研究、交叉勘探策略、并发系统的覆盖标准、突变分析、监测和调试方法、静态分析、模型检查，以及编程范例。我们知道，这些选择可能会使已定义的框架产生偏差，并可能使某些社区的工作优于其他社区。通过明确列出我们调查中排除的工作，我们公平地指出了我们工作的界限和限制。

15年来，我们调查的90多个出版物中包含了大量的出版物，这些出版物的社区和场所的异质性使我们对分析的有效性充满信心。

7项相关工作

我们将本次调查的范围界定为针对并发性故障的软件测试技术，并有意忽略并发系统的其他方面。在本节中，我们简要概述了本次调查范围边界上的一些重要领域：在第7.1节中，我们概述了支持并行软件系统测试方法的研究和技术，以及与本次调查框架特别相关的研究和技术；在第7.2节中，我们概述了Veri的主要方法。在第7.3节中，我们提出了可替代本文所述测试技术的并发系统，以简化并发系统的开发。

7.1相关研究

本文所研究的软件测试技术受并发故障的实证研究的影响很大，利用不同的策略来探索交织状态，并对并发软件系统定义了补充覆盖标准。

7.1.1并发故障的实证研究

对于正在开发和野外并发断层的大型实证研究结果，很少有论文报道，但它们强烈影响了本次调查中提出的一些技术。经验研究和测试技术之间最直接的联系是观察到，只有两个执行流出现大量并发故障，这强烈影响了许多为并发系统生成测试用例的技术。

对并发故障的第一个全面和广泛引用的研究是由Lu等人提出的。[130]考虑从四个开放源代码应用程序中随机选择105个并发错误。以前的研究，如Chandra和Chen[141]对少量并发故障进行了研究，限制了结果的可推广性。Lu等人的实验结果引出了一些有趣的考虑：（i）大约三分之一的非死锁并发故障是违反顺序假设的；（i i）大约三分之一的非死锁并发故障涉及多个变量；（i i i）大约92%的并发故障可以通过强制执行特定的顺序可靠地触发。在最多四个内存访问中；（iv）大约73%的非死锁并发故障不是通过简单地添加或更改锁来修复的，并且很多时候第一个修复是不正确的，这表明对并发系统进行推理的困难；（v）可能导致程序崩溃的并发故障约占非死锁并发故障的50%。-死锁故障；（vi）许多众所周知的交织属性，如数据争用和原子性冲突，与故障严重性几乎没有关系；（vii）大约85%的并发故障导致的崩溃涉及对共享内存对象的错误内存访问；（viii）并发内存故障可以进一步分为空sha红色指针取消引用、共享缓冲区溢出、对共享数据项的未初始化读取和指向共享内存的悬空指针。

丰塞卡等人。[142]研究MySQL应用环境中的并发错误，观察到15%的并发错误不会导致死锁或系统错误，而是悄悄地破坏某些数据结构，并违反应用程序的预期语义。林和DIG（143）在Java中对并发集合的误用进行了实证研究，特别关注在错误的假设条件下开发代码时发生的检查行为错误，以及依赖于这种评估结果的后续操作。作者考虑了不同类型集合的不同使用模式，并观察到一些模式具有很高的暴露并发性错误的概率，这表明它们是一个很好的测试目标。

7.1.2基准

只有很少的研究关注于评估并行软件系统测试方法的基准。林等人。[144]回顾该领域基准的现状，并观察到：（i）研究人员通常遵循特定的过程来选择案例研究进行评估，以及（i i）现有的基准没有明确设计用于并发故障检测，通常过时，由可能引入偏差的现实案例研究选择组成。当评估测试技术的有效性时。在他们的调查中，Lin等人介绍8个软件项目中47个故障的公开基准jacontebe，这可能是研究工作的参考基准。7.1.3交错勘探策略

交错空间随执行流的数量和每个执行流中的操作数呈指数增长。探索空间的问题是通过减少或限制待探索空间的算法来解决的。

减少交织空间的方法是利用目标系统的已知特性对搜索空间进行删减。Java路径查找器〔131〕，〔145〕介绍了偏序约简，它利用独立操作的交换性来避免导致相同结果的交织的重复分析。部分顺序减少[32]，[146]消除了勘探过程中的等效交错，从而限制了分析过程中的内存消耗。在最佳情况下，还原级别可以是搜索空间大小的指数级。更具侵略性的方法进一步限制了某些特定交叉的执行：例如，K€Ahkonen等人的方法[147]通过将动态符号执行与目标系统的Petri网模型的展开相结合，来交换性能的完整性。

虽然减少交织空间的方法旨在完全覆盖搜索空间，但结合交织空间的方法确保了一定范围内勘探的完整性。在顺序程序分析中，通常采用深度边界，以将执行空间的探索限制在有限的步数范围内[148]。在并发程序领域，Chess模型检查器引入了上下文边界，限制了在分析期间要考虑的最大上下文切换数[31]、[149]、[150]。Delay Bounding[151]Bounds计划可以偏离预定义的确定性计划程序的次数。汤姆森等人对延迟和上下文边界的经验比较表明，上下文边界并没有引入明显的改善，而延迟边界在寻找断层方面更有效[152]。汤姆森等。请注意，在有限的范围内可以发现许多错误；但是，大多数并发错误不会导致立即的系统崩溃，而是导致更细微的错误，例如数据损坏。

宾达尔等。[153]提出变量和线程边界来进行排序和探索交错的空间。它们从观察到的实际并发性错误通常涉及少量的共享变量和执行流，并将这些数字用作边界的观点出发。Reex[154]关注回归测试，并优先考虑可能受测试系统变化影响的交错。

7.1.4并发系统覆盖标准

第5.6节（表6中的行覆盖标准）中介绍的一些空间探索技术根据并发系统的覆盖标准确定了要探索的程序空间子集，这些标准也独立于测试方法进行研究。

布隆等。[155]提出同步覆盖范围，涉及同步机制。该标准建立在可用并发机制的模型和常见故障模式的知识之上。例如，作者讨论了Java提供的并发抽象的同步覆盖的设计。陆等。[156]提出一套覆盖标准，讨论其成本，范围从指数o与访问共享数据项的次数成线性关系，根据子项关系分层组织标准，并讨论每个标准处理的故障类型。Krena等人。[157]提出一种从目标错误模式开始构建覆盖率标准的通用方法，重点关注适合指导基于搜索的技术的标准，如遗传算法。洪等。[158]研究一些覆盖标准对测试有效性的影响，观察到这些标准是同时测试有效性的中度到良好的预测因子，并且通常是测试用例生成的合理目标。

7.2并发软件系统的验证与确认

本调查中讨论的测试技术是并发软件系统的验证与确认方法之一，包括突变分析、运行时监视与调试、静态分析、模型检查和模型一致性验证。

7.2.1并发系统突变分析

突变分析通过对通过系统地播种程序更改获得的一组突变程序执行该套件，并对结果进行统计分析，来估计目标程序的测试套件的有效性。突变分析将一组突变算子应用于目标程序，以生成系统的修改版本，称为突变体，对每个突变体执行测试套件，并检查突变体是否可以被杀死，也就是说，是否与原始程序不同。被杀突变体相对于（非等效）突变体总数的比率是估计测试套件质量的突变分数。感兴趣的读者可以参考贾和哈曼最近关于突变检测的优秀调查[159]。突变分析已经扩展到并发系统，方法是：（i）定义并发系统的突变算子；（i i）引入将突变分析应用到并发系统的技术和工具。

Bradbury等人〔160〕定义了一个自5版本以来在Java中引入的并发和同步机制的一组变异算子。Bernhard和Delgado[161]建议通过在规范级别上应用突变算子为并发系统生成突变体。sen和abadir[162]为并发实时系统提供了突变算子。

将突变分析应用于并发系统时，出现的一个相关问题是将执行原始程序产生的结果与执行突变体产生的结果进行比较。原则上，应评估所有可能的操作交错，以确定结果的差异。卡弗的开创性著作[163]提出了一种启发式方法，只考虑所有可能的交错的子集。mutmut框架[164]利用在被测原始代码执行过程中收集的信息，加快对突变代码中相同测试的探索和执行。尤其是，mutmut修剪不受所考虑的突变影响的交错。

Gligoric等人[165]通过提出一种选择性突变技术来解决大量突变体的问题，该技术可识别出高效且节省成本的操作子集。选择性突变产生的突变体较少，接近突变分数。

Aichernig等人[161]修改规范并生成测试用例，检查系统是否符合故障。规范。aichernig等人的一些突变操作符与并发性严格相关，例如交换两个事件的顺序，或者用另一个消息替换消息。

7.2.2并发系统的监视和调试

监控技术收集正在运行的系统的信息，以了解执行是否违反了某些所需的属性，并提供复制失败执行的信息。监控并发系统的方法，特别是复杂的分布式系统，可以在线或离线工作。在线监控方法在执行期间分析目标系统，而离线方法离线分析日志记录和检查点信息。

喜鹊[166]在运行时收集性能跟踪，并使用它们构建一个可以离线分析的性能模型。喜鹊是基于一个黑盒仪器，不需要修改源代码的测量系统。x-trace[167]标识协议栈不同层上任务之间的脱机关联。它使用元数据来增加任务，并强制跨层传播此类元数据，以提供分布式应用程序行为的统一视图。PIP[168]提出了一种语言来为分布式系统指定所需的属性，并检查这些属性在运行时是否满足。星期五[169]和WIDS检查器[170]重放（失败）分布式系统的执行以支持调试。反序列化程序[171]侧重于性能监控，并使用机器学习技术检测最可能导致性能问题的一组事件。辛格等。[172]通过查询系统状态，在线监控分布式系统。d3s[173]和maceodb[174]检查联机全局并发属性。CrystalBall[175]，[176]进一步预测了部署系统中的潜在故障，并试图防止它们发生，从而使执行远离违反安全特性的状态。

7.2.3并发系统静态分析

静态分析方法的目的是通过依赖静态可用信息（即代码或某种形式的规范）来验证系统的某些期望属性，而不需要执行任何系统。静态分析技术针对的是与本文所调查的测试方法不同的完整性-精度权衡。静态分析通过处理整个程序空间来提高完整性，但由于错误的正警告（源自保守的近似值，可能导致不可行的交错），可能会导致精度不足。

Racerx[177]是Dawson为检测数据争用和死锁而进行的静态分析的开创性工作，它依赖于对目标系统控制流图的静态锁集分析，并通过一种估计检测到的故障严重性的机制来限制误报问题。von Praun和Gross[178]利用静态分析检测原子性违规。它们建立在方法一致性的概念之上，该概念检查方法范围内的访问是否与其他方法的冲突访问交错。von praun和gross对程序进行静态分析，以推断每个方法的锁定规则和内存访问模式的信息，如果违反方法一致性，则报告原子性违规。Williams等人[179]静态分析代码，构建库的锁序图，检测对库公共方法的调用序列当并发执行时会导致死锁。Jade[180]通过检查六个表示死锁必要条件的属性，静态计算锁获取图并减少误报数量。马里诺等人。[181]使用同步原语实现原子集可序列化的目标程序。它们静态地在原子集上构建部分锁获取顺序，并在找不到部分顺序时生成警报，表示潜在的死锁。

7.2.4模型检查

模型检查是一种正式验证有限状态并发系统的方法[182]。模型检查是80年代初由艾默生和克拉克[183]以及奎尔和西法克斯[184]提出的，它包括详尽地遍历系统的抽象模型，并检查是否存在某些用某种逻辑形式主义表示的感兴趣的属性。

在本次调查中，我们将模型检查与锁集分析[29]、[30]、[43]相结合，回顾了几种依赖模型检查来生成并发系统的测试用例和交织的方法，这些方法发生在分析[66]、[67]混合分析[103]、详尽探索[125]、[126]、[127]、[128]、[129]和启发式分析[137]之前。

模型检查在并发软件系统分析中也得到了广泛的应用，并且与其他静态分析技术的优点和局限性相同。模型检验的许多应用的全面调查不在本次调查范围之内。感兴趣的读者可以参考克拉克的书[182]和关于模型检查的许多研讨会的会议记录。

7.2.5模型一致性

模型一致性方法从被测系统的一些抽象模型中生成测试用例，以验证系统是否符合模型。

在本次调查中，我们回顾了产生具体测试用例的模型一致性方法，特别是针对原子性违规的技术，这些技术引入了原子代码块的规范，并检查实现是否确保了这些块的原子性[27]、[35]、[36]。这里，我们简要描述了在具体测试技术中没有实现的模型一致性方法，因此不在本次调查的范围内。模型一致性可以追溯到对协议一致性检验的研究[185]，并且已经扩展到许多形式和符号：多端口有限状态机[186]、输入/输出转换系统（IOTS）[187]、[188]、部分阶自动机[189]、[190]和Petri网[191]。即使部分顺序自动机和Petri网嵌入了简化并发系统规范的并发符号，该领域的大多数工作都集中在多端口FSM上，其中分布式系统的组件被建模为端口，对一个端口的输入可以触发在多个端口产生输出的转换。

7.3并发性编程范例几个编程范例被提议开发并发软件系统和表达并发模式。成功的编程范例有：OpenMP[192]中的数据并行和任务并行原语、苹果的中央调度和操作队列[193]、编程事件循环[194]、微软的任务并行库。在.NET（195）中，异步编程的原语，如异步调用、期货和承诺（196），以及Java 5以来引入的Java并发实用程序。在这里，我们简要地总结了不同的编程范式是如何防止某些并发故障的发生，还是如何在编译时实现有效的静态检测。

Haskell[197]等函数式编程语言不允许可变状态，因此通过设计保证了种族自由。因此，大多数用于大数据并行和分布式处理的现代框架（如MapReduce[198]、Apache Spark[199]和Apache Flink[200]）都提供了功能接口。类似地，一些编程语言和库提供操作集合的函数抽象。这是ReActux框架（201）和Java 8中引入的流运算符的情况。基于参与者的语言，如Erlang[13]、[202]和scala/akka[14]使用消息实现执行流之间的通信，并确保消息被原子地处理。由于两个执行流不能同时访问共享资源，因此通过设计，这可以保证数据争用的自由。

一些类型系统通过设计来约束并发模式以确保某些给定属性的有效性：一些类型系统避免死锁[203]、[204]、其他类型系统避免数据争用[205]、[206]、[207]而其他类型系统则避免原子性冲突[208]。面向对象编程框架Scoop为并发编程提供了高级原语，以确保数据争用自由，排除某些形式的死锁，并在执行流之间实现前置和后置条件推理保证[209]。任务并行的twejava[210]模型允许声明每个任务对共享状态的影响。运行时确保只有在两个任务的效果不冲突的情况下，它们才会同时执行。P[211]扩展了C用于基于事件的编程，并强制声明性规范与状态机并发，从而实现良好的静态数据竞争分析。

其他编程范例和类型系统确保了确定性的执行顺序。确定性并行Java（DPJ）[212 ]引入了一种类型和效果系统，该系统通过Java中的编译时间检查来保证确定性语义。Grace（213）是一个C/C++程序的运行时系统，它屏蔽了在多核上顺序设计的代码的并发执行。CIEDET（214）和D螺纹（215）取代了C/C++程序的P螺纹库，提供了有限的开销的确定性执行。确定性程序[216]为微内核之上的低开销确定性并发执行提供了API。

8结论

本文综述了并行系统测试的最新结果。这一领域的研究主要集中在选择待测交错区的问题上，并探索了两类方法：基于属性的方法，其目标是更有可能导致断层的交错区模式；以及基于Coverag的勘探方法，其目标是详尽地探索交错区的空间。标准或启发式。

目前的研究趋势是基于预测性属性的技术和违反预期的顺序不变量，而不是低级别的内存访问冲突，如数据竞争。过去十年的研究为并行系统提供了几种有效的测试技术，为未来的研究开辟了广阔的方向：

（i）大多数并发系统的测试技术都以选择相关的交错为目标，很少有技术专注于测试用例的生成。利用这两个方面之间的协同作用仍然是一个开放的研究课题。

（ii）绝大多数测试方法都针对共享内存系统。分布式消息传递系统的验证和验证主要采用静态分析和模型检查的方法，使测试消息传递系统的重要领域开放，以供将来的研究。

（iii）在过去的十年中，并行软件系统出现了大量新的编程范例，这些范例强制执行流之间的交互模式，以防止出现诸如数据争用和死锁之类的并发错误。新的编程范式将测试问题从低级内存访问冲突转变为高级顺序冲突，并为开发利用现代编程范式语义的新测试方法提供了机会。