一种验证分布式协议活性属性容错机制的模型检测方法

陆超逸[1)] 聂长海[1)] 张成志[2)]

[1)](南京大学计算机软件新技术国家重点实验室 南京 210023)

[2)](香港科技大学 香港 999077)

摘 要 云计算是一种通过网络以服务的方式向用户提供按需收费的计算资源的模式,目前企业逐渐将业务部署、数据处理转移到云计算平台上进行.因为可扩展性、性能等各方面需求,所以云平台部署在分布式系统上.由于分布式系统采用大量的商品机通过复杂的结构进行搭建,因此分布式系统中组件发生故障是无法避免的.为了提高分布式系统的可靠性,技术人员在开发分布式系统时为其设计了容错机制.为了保证容错机制在分布式系统发生故障时能真正有效地工作,故障注入是检验容错机制的方法之一,通过人为地向系统中注入特定的故障,观察系统的行为并检验容错机制是否正确工作.由于分布式系统的并发特性,传统软件测试方法无法对其进行完全测试,近年来越来越多地使用模型检测技术来对分布式系统进行验证.现有的模型检测技术注重对分布式系统的安全性属性和活性属性的检测,忽略了对容错机制尤其是活性属性容错机制的检测,所以如何验证系统的活性属性容错机制是目前面临的挑战.采用抽象模型检测方法会引入模型与实际系统不匹配的问题.同时,采用实现级模型检测方法会加剧模型检测中的状态空间爆炸问题.本文提出了一个实现级模型检测工具LTMC(Liveness Properties Fault Tolerance Model Checker),结合故障注入技术对分布式协议的安全性属性与活性属性及其容错机制进行验证.同时,基于分布式系统节点的角色,本文提出了一种对等约减策略PRP(Peer Reduction Policy)对LTMC需要搜索的状态空间进行约减,缓解了状态空间爆炸问题.此外,LTMC通过引入逻辑时钟机制,优先搜索那些更有实际价值的事件执行路径.LTMC能够有目标地在待验证系统运行的特定时刻注入特定的故障,而不依赖于随机故障注入策略;当待验证系统发生改变时,只需要简单地对工具进行轻微的修改;LTMC可以系统地发现分布式协议中指定类型的所有Bug.在本文最后,我们将LTMC应用到ZooKeeper和Cassandra的几个协议中,并与深度优先搜索作对比,可以发现LTMC有3.7~594.4倍的状态空间约减率.

关键词 分布式系统;模型检测;故障注入;活性属性;容错机制;对等约减策略

中图法分类号 TP311 **DOI号** 10.11897/SP.J.1016.2021.01714

A Model Checking Method for Verifying the Fault Tolerance of Distributed Protocol Liveness Properties

LU Chao-Yi[1)] NIE Chang-Hai[1)] Shing Chi CHEUNG[2)]

[1)](*State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing* 210023)

[2)](*The Hong Kong University of Science and Technology, Hong Kong* 999077)

Abstract Cloud computing is a new mode that provides users with computing resources charged on demand through a communication network as a service. At present, enterprises are gradually transferring business deployment and data processing to cloud computing platforms. Because of various requirements such as scalability and performance, cloud platforms are deployed on distributed systems. Since the distributed system uses a large number of commodity machines to build through a complex structure, technicians cannot guarantee that these machines will work

收稿日期:2020-03-04;在线发布日期:2021-01-20.本课题得到国家重点研发计划(2018YFB1003800)资助.**陆超逸**,硕士,主要研究方向为模型检测与软件测试.E-mail:DZ1733010@smail.nju.edu.cn.**聂长海**(通信作者),博士,教授,博士生导师,主要研究领域为软件测试与软件质量保证.E-mail:changhainie@nju.edu.cn.**张成志**,博士,教授,博士生导师,主要研究领域为软件工程.

correctly all the time. Therefore, component failures in the distributed system are unavoidable. In order to improve the reliability of the distributed system, the technicians designed fault-tolerant mechanisms for them when developing the distributed system. In order to ensure that the fault-tolerant mechanisms can really work correctly when there is a failure in the distributed system, fault injection is one of the most effective ways to test the fault-tolerant mechanism, observing the behavior of the system and verifying whether the fault-tolerance mechanism is working correctly, by artificially injecting specific faults into the system under test. Due to the concurrent nature of distributed systems, it is very difficult for traditional software testing methods to fully test such systems. In recent years, technicians have increasingly used more systematic model checking techniques to verify distributed systems. However, existing model checking technology focuses on the checking of the safety properties and liveness properties of distributed systems, and ignores the checking of fault-tolerant mechanisms, especially liveness properties fault-tolerant mechanisms. As a result, how to verify the liveness properties fault-tolerant mechanism of the distributed system is currently a challenge. The use of abstract model checking method will introduce the problem of mismatch between the model and the actual system because of human error. At the same time, the use of implementation-level model checking method will aggravate the state space explosion problem. As a result, in this paper we propose an implementation-level model checking tool LTMC (Liveness Properties Fault Tolerance Model Checker), which combines fault injection technology to verify the liveness properties with their fault tolerance mechanisms and safety properties of the distributed protocols in the distributed systems. At the same time, based on the role of distributed system nodes, this paper proposes a PRP strategy (Peer Reduction Policy) to reduce the state space that LTMC needs to search, alleviating the problem of space explosion. LTMC can purposefully inject specific faults at specific moments when the system to be verified is running, instead of relying on random fault injection strategies; when the system to be verified changes, only slight modifications to the model checker are required; LTMC can systematically discover all bugs of the specified types in the distributed protocols. In addition, LTMC prioritizes the exploration of more practical execution paths of events by introducing a logic clock mechanism. At the end of this article, we apply LTMC to several protocols of ZooKeeper and Cassandra which are the most popular distributed systems on the market, and compare LTMC with depth-first exploration. LTMC has a state space reduction rate of 3.7—594.4 times.

Keywords　distributed system; model checking; fault injection; liveness properties; fault-tolerance mechanisms; peer reduction policy

1　引　言

如今随着大数据技术日渐成熟,数据规模激增,使用本地单一机器逐渐不能满足对数据处理的需求,所以越来越多的数据和服务都被移动到云端进行存储和计算,云端使用分布式系统在机器集群间进行信息的交互,协调多个机器共同完成任务.目前有很多分布式系统提供各种服务,如 ZooKeeper、Hadoop、Berkeley DB 等.这些分布式系统提供的服务改变了原始的软件架构,客户端应用变得轻量化,软件更多地依赖于服务,因此对分布式系统的正确性和容错性提出了严格要求.

由于分布式系统协议涉及大量的并发操作,这就存在许多由并发而产生的不确定性,使得技术人员保证系统的正确性十分困难,并且由于这种不确定性使得系统错误的重现几乎变得不可能.例如,在同一时刻系统中有四个并发事件分别用 a、b、c 和 d 来表示,那么系统对这四个事件的处理顺序便有 $4!=24$ 种:$abcd$、$abdc$、$acbd$ 等,这 24 种事件执行序

列可能会对系统造成不同的结果，要想验证系统行为满足某些性质，那么要将系统驱动到所有可能的状态，对每个状态分别进行验证，这就需要对系统中所有的并发事件（本例中的 $abcd$ 四个事件）进行重排序，从而获得系统所有可能的执行事件的序列，执行这些事件序列就能遍历系统所有可达的状态，对每一个可达的系统状态进行验证.

同时，由于整个分布式系统构架在商品机集群上，无法保证所有节点每时每刻都能正常工作，例如在 2017 年，亚马逊位于美国北弗吉尼亚的云存储服务器出现故障，导致使用该服务器的数千个网页完全无法访问，大量 APP 功能失效，十多万个网页内部分链接失效且图片无法显示①. 因此为分布式系统设计容错机制并保证其正确工作非常重要，容错机制保障系统在存在个别节点故障的情况下也能够继续准确无误地完成任务. 目前分布式系统广泛使用备份节点的方法来实现系统的容错[1]，那么如何验证系统的容错机制是否能够正确工作？如果等到系统真正出现故障再检查容错机制是否正确执行显然是不实际的，所以可以采用故障注入技术，主动地向目标系统中注入故障，验证被注入故障的系统是否依旧能够正确地完成任务.

使用传统的软件测试方法无法对分布式系统的正确性和容错性进行完全保证. 由于分布式系统存在并发不确定性，导致即使使用同一条测试用例进行多次测试，也无法保证将系统对该测试用例所有可能的行为都能测试完备. 而且，软件测试的目的在于更快地找出待测系统中的 Bug，而不是证明待测系统不存在 Bug. 所以，一个分布式系统即使通过测试，也无法说明该分布式系统中所有的 Bug 都被找出或者不存在 Bug.

形式化方法可以对分布式系统行为的所有可能性都进行验证，弥补了软件测试在这个方面的不足，从而系统地找出分布式系统中所有的 Bug 或者证明不存在 Bug. 目前保证分布式系统的正确性和容错性的形式化方法有多种，其中模型检测是最有效的方法之一[2-3]，并且模型检测已经成功地在工业界得到应用，如硬件电路系统的验证[4]和软件系统的验证[5-7]. 在硬件电路设计中，一个微小的问题可能会使所有制作的产品报废，造成巨大的成本损失；在一些安全攸关的软件系统中，Bug 的出现会导致严重的生命财产损失，例如航天系统、高铁系统等；在一些高并发软件系统中，例如分布式系统，存在许多不确定性，这些不确定性使得软件测试技术不能进行完备的测试. 在上述软硬件系统中采用模型检测可以帮助技术人员解决软件测试的完备性问题，增加对系统正确性的信心，从而降低后续造成损失的风险. 与形式化方法中的另一类方法定理证明比较，定理证明需要技术人员在前期对待验证系统的语句进行标注，而模型检测技术只需要对系统进行建模便可以自动化地进行验证. 对于规模较大的分布式系统，使用定理证明成本过高，所以使用模型检测技术是最合适的选择. 模型检测技术对待验证的系统进行建模，充分考虑目标系统所有可能的行为，对每一种可能的行为进行验证，检验其是否满足规定的性质[8]，其工作机制如图 1 所示. 在模型检测中，有两种属性需要去验证：安全性属性和活性属性[9]. 安全性属性是指，在系统每一个状态都必须满足的属性，即坏事情永远不会发生，例如系统不存在死锁就是一个安全性属性；活性属性是指不需要在系统的所有状态都满足，但总在未来的某个状态下满足的属性，即好事情最终会发生，例如在需要选举 Leader 的分布式系统中，系统不总是存在 Leader 的，但经过一系列 Leader 选举过程，最终系统会产生 Leader，所以系统存在 Leader 是这种类型系统的活性属性. 下面给出安全性属性 φ 和活性属性 ψ 的定义.

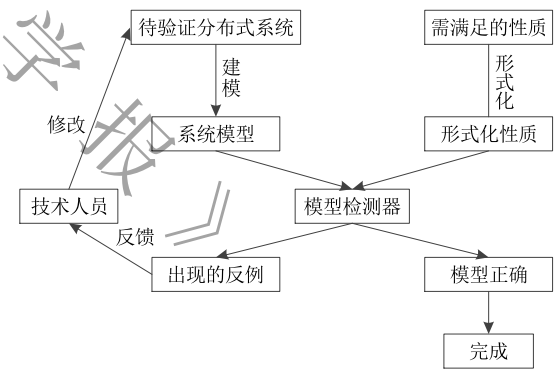


图 1 模型检测流程

定义 1. 安全性属性 φ. 对于系统的每一条执行路径 π，$\forall i \geq 0$，π_i 满足 φ，其中 π_i 表示路径 π 上的第 i 个状态.

定义 2. 活性属性 ψ. 对于系统的每一条执行路径 π，$\exists i \geq 0$，π_i 满足 ψ，其中 π_i 表示路径 π 上的第 i 个状态.

模型检测技术分为实现级模型检测（implementation-level）[2-5,9]和抽象模型检测[10-11]，实现级模型检测是直接在待验证系统的源代码进行运行时验证；而抽象模型检测是将待验证系统转换成抽象

① https://tech.huanqiu.com/article/9CaKrnK0VVt

模型(自动机[12]、Petri 网[13-14]等),在抽象模型上使用图论、集合论等相关知识进行验证.使用抽象模型的模型检测只能验证该系统的设计以及其抽象模型,却无法发现由于代码实现而引入的 Bug;同时,对待验证系统进行抽象建模可能会引入模型与系统不匹配的问题,在不匹配的模型上进行模型检测是毫无意义的.因此本文使用实现级模型检测来对分布式系统协议进行验证.实现级模型检测直接运行大规模的软件源码,虽然这会比抽象模型检测更容易导致状态空间爆炸的问题,但是通过黑盒[9]、白盒[15-16]的空间约减策略,状态空间爆炸问题会得到缓解.黑盒空间约减方法具有强移植性,但是不能根据待验证系统的特性对状态空间进一步约减;相反,白盒空间约减方法可以分析待验证系统特有的性质,利用这些性质能够得到更好的约减效果,但出于商业机密等原因,想要拿到系统源码进行分析往往存在困难.

然而,现有的模型检测工具都将重点聚焦于分布式系统安全性属性和活性属性的验证,忽略了对容错机制尤其是活性属性容错机制的检测,因此,验证分布式系统的容错机制尤其是活性属性容错机制是目前面临的挑战,也是本文需要解决的主要问题.虽然,目前一些模型检测工具支持向待验证系统中注入故障来验证容错性,但这些模型检测工具注入故障往往是随机注入或者穷举故障可能出现的所有情况(在待验证系统运行的任意时刻都注入故障)[2,10-11,15-16],这样做往往对故障注入点的选择缺乏目的性,带来很多问题和挑战,例如使用随机注入无法找出所有容错机制不能正常工作的地方,而穷举注入会加重空间爆炸的问题,使得模型检测器不具备扩展性.本文将模型检测和故障注入技术相结合,提出了一种有目的地选择故障注入点的方法和一种基于分布式系统节点角色的对等约减策略 PRP(Peer Reduction Policy),实现了相应的工具 LTMC(Liveness properties fault Tolerance Model Checker),对分布式系统的正确性和活性属性的容错机制进行检测.

本文的主要贡献点如下:

(1)提出了一个将故障注入与模型检测相结合的实现级(implementation-level)模型检测工具 LTMC,检查待验证系统的安全性属性和活性属性是否满足,并对活性属性的容错性进行验证.

(2)LTMC 不使用随机注入或者穷举注入的故障注入方法,而是有目的地将特定的故障在特定时间注入到待验证系统的指定位置,这种故障注入方法既避免了随机注入的偶然性,相比于穷举注入又在一定程度上约减了由于注入故障而引入的状态空间.

(3)提出了一种基于节点角色对等的状态空间约减策略 PRP.该策略能够大大减少 LTMC 需要搜索的事件执行序列,使得 LTMC 具有实用性.

本文第 2 节介绍背景知识;第 3 节提出我们的方法,并进行案例分析;第 4 节展示根据该方法所实现的原型工具,并且通过实验证明了所提方法的有效性;第 5 节介绍了相关工作;第 6 节总结本文工作并提出未来的研究方向.

2　背景知识

2.1　计算树逻辑 CTL

计算树逻辑(Computation Tree Logic)是一种分支时间逻辑,将系统运行轨迹建模成树状结构,其中每个状态可以有几种不同的后续状态,CTL 能够对系统从某个状态开始的运行轨迹上需要满足的性质进行描述.

定义 3.　CTL 的语法如下:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid A\psi \mid E\psi$$

$$\psi ::= X\varphi \mid F\varphi \mid G\varphi \mid \varphi_1 U \varphi_2 \mid \varphi_1 R \varphi_2$$

其中:

p 代表一个原子命题;

A 代表对于所有路径满足 ψ;

E 代表存在某些路径满足 ψ;

X 代表当前状态的下一个状态满足 φ;

F 代表将来,说明在路径上将来的某个状态满足 φ;

G 代表总是,说明在路径上所有状态满足 φ;

U 代表直到,说明路径上某个状态满足 φ_2,且此状态之前的所有状态满足 φ_1;

R 代表释放,说明路径上从初始状态开始到包括第一个满足 φ_1 的状态结束,φ_2 一直满足.

下面介绍 CTL 逻辑的语义,令 f 为使用 CTL 描述的公式,M 表示待验证的系统,s 表示系统中的一个状态,那么 $M,s \models f$ 表示公式 f 在系统 M 的状态 s 处满足.用 π 来表示系统运行过程中的一条执行路径,π^i 表示该路径上的第 i 个状态(路径 π 的初始状态为第 0 个状态),那么满足关系 $\models$ 可以通过下面规则递归定义.

(1)$M,s \models p \Leftrightarrow$ 在状态 s 处原子命题 p 成立.

(2) $M,s\models\neg f\Leftrightarrow M,s\not\models f$.

(3) $M,s\models f_1\vee f_2\Leftrightarrow M,s\models f_1$ 或者 $M,s\models f_2$.

(4) $M,s\models f_1\wedge f_2\Leftrightarrow M,s\models f_1$ 并且 $M,s\models f_2$.

(5) $M,s\models Ef\Leftrightarrow$ 存在一条从 s 出发的路径 π 满足 f,即 $M,\pi\models f$.

(6) $M,s\models Af\Leftrightarrow$ 所有从 s 出发的路径 π 都满足 f,即 $M,\pi\models f$.

(7) $M,\pi\models f\Leftrightarrow s$ 是路径 π 上的第一个状态,并且 $M,s\models f$.

(8) $M,\pi\models Xf\Leftrightarrow M,\pi^1\models f$.

(9) $M,\pi\models Ff\Leftrightarrow$ 路径 π 上存在一个状态 s 满足 $M,s\models f$.

(10) $M,\pi\models Gf\Leftrightarrow$ 对于路径 π 上的每一个状态 s 有 $M,s\models f$.

(11) $M,\pi\models f_1Uf_2\Leftrightarrow$ 存在 $k\geq 0$ 满足 $M,\pi^k\models f_2$,并且对于所有 $0\leq j<k$ 满足 $M,\pi^j\models f_1$.

(12) $M,\pi\models f_1Rf_2\Leftrightarrow$ 存在 $k\geq 0$ 满足 $M,\pi^k\models f_1$,且对于 $0\leq j<k$ 满足 $M,\pi^j\not\models f_1$,同时 $M,\pi^j\models f_2$.

2.2 公平性

公平性(fairness)可以对并发程序的进程执行进行限制,公平性原则用于排除一些系统不合理的进程交错执行路径.有两种公平性原则:

(1)强公平性假设.无限次被经常使能的进程,要求必须被无限次经常执行.

(2)弱公平性假设.一直被使能的进程,要求必须被无限次经常执行.

2.3 分布式系统模型检测

分布式系统是由多个独立的节点所构成的完整系统,每一个节点都需要完成各自的任务,同时,节点之间需要通过相互协作完成整体的目标.节点之间的相互协作是通过信息交互完成的,其主要方式有:消息传递、共享资源访问等.节点在一个时刻可能执行的事件有多种类型,例如向其他节点发送消息、收到其他节点发送的消息并处理、节点崩溃、节点恢复等.前面两种事件是由分布式协议的逻辑来触发的,而后面的节点崩溃和节点恢复事件是由于系统中存在的某些故障造成的,是技术人员事先不可预知的.

2.3.1 状态与状态迁移

在分布式系统中,节点每执行一个事件都会改变系统的状态,会将系统驱动到一个新的状态,本文将节点执行一个事件称为一次系统状态的迁移,连续的迁移称为迁移序列,定义 4 和定义 5 给出迁移和迁移序列的定义.如图 2 所示,系统在状态 s_0 下通过执行 t_0 对应的事件到达状态 s_1,t_0 即为状态 s_0 下的迁移;系统在状态 s_0 下通过连续执行迁移 $t_0t_1\cdots t_{n-1}$ 到达状态 s_n,那么 $t_0t_1\cdots t_{n-1}$ 就是状态 s_0 下的一条迁移序列.

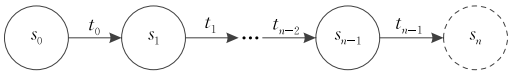


图 2 系统状态迁移

定义 4. t 是系统在状态 s 下的一个迁移,当且仅当在状态 s 下系统可以执行 t 所对应的事件并且 $s\xrightarrow{t}s'$,其中 s' 是系统的一个状态,s' 与 s 可以是相同状态.

定义 5. $T(t_0t_1\cdots t_{n-1})$ 是系统在状态 s_0 下的一条迁移序列,当且仅当在状态 s_0 下存在一系列后续状态使得 $s_0\xrightarrow{t_0}s_1\xrightarrow{t_1}s_2\cdots\xrightarrow{t_{n-1}}s_n$,其中 $t_i(i=0,1,2,\cdots,n-1)$ 是系统的迁移,$s_i(i=0,1,\cdots,n)$ 是系统的状态,s_i 之间可以是相同状态.

2.3.2 状态搜索

由于分布式系统内部存在大量并发事件,导致这些事件的发生顺序存在不确定性,所以实现级模型检测要将并发事件进行重排序,搜索出待验证系统中所有可能的节点事件执行顺序.即在初始状态下搜索出所有的迁移序列,并执行这些迁移序列对应的事件,这样就能够遍历待验证系统的状态空间.状态空间的搜索方法有如下两种:

(1)随机搜索.基于随机的状态搜索策略将待验证系统从初始状态开始,每次随机选择一个当前可执行的事件执行,直到事件序列长度达到 k,那么一条随机搜索的事件序列就得出来了.根据技术人员的要求,重复上述操作得出 n 条随机搜索的事件序列.

(2)深度优先搜索和广度优先搜索(DFS 和 BFS).对于 DFS 深度优先搜索,每次只选择该状态下的一个动作进行搜索,当达到最大深度之后再回溯到先前的状态来搜索其他动作;对于 BFS 广度优先搜索,每次到达一个系统状态时将该状态下的所有动作全部搜索,之后进行下一深度的搜索直到到达最大的深度.

可以发现,随机搜索策略不需要搜索出所有事件执行序列,只需要每次随机选择一个可以执行的事件进行执行,所以该策略的负载小,但是不能对系统的状态空间进行遍历,所以该状态搜索方法可以用于软件仿真.DFS 和 BFS 都能够穷尽所有指定深度的所有搜索序列,但是 DFS 比 BFS 的优势在于

不需要频繁的状态切换,而 BFS 由于在一个状态 s 下要执行该状态下所有可能的动作,所以在每次执行一个动作后都要恢复到 s,这就需要频繁的状态切换,该过程开销较大.因此在 LTMC 中使用的状态空间搜索方法都是基于 DFS 考虑的.

2.3.3　状态空间约减

然而,随着需要重排序的并发事件数量的增加,模型检测器搜索并执行的事件序列数量呈指数级增长.此时,需要对事件执行序列进行约减.动态偏序约减(DPOR)[17-18]是常用的约减事件序列数量的方法之一.偏序约减[17]给出两个并发事件 a、b 的独立性定义:

定义 6.　如果事件 a 和 b 同时满足下面的条件,那么称 a 与 b 相互独立:

(1)存在一个状态 s,如果 $a,b\in enable(s)$,并且满足 $b\in enable(a(s))$ 与 $a\in enable(b(s))$.其中 $enable(s)$ 表示状态 s 下可以执行的事件,$a(s)$ 表示在状态 s 执行事件 a 后到达的状态.

(2)存在一个唯一的状态 s' 使得 $s\xrightarrow{ab}s'$ 且 $s\xrightarrow{ba}s'$.

从定义 6 可以得出,交换两个的独立事件,会到达相同的状态;反之,如果两个并发事件 a、b 不独立,那么交换事件 a 和 b 的执行顺序,系统会被驱动到不同的状态.

DPOR 的具体做法:在当前状态 s 下搜索出一个动作 a 时,检查在状态 s 之前是否存在一个最近的状态 s',在该状态下也能执行动作 a,并且在 s' 下执行的动作是 a',如果 a' 和 a 不独立且 a' 和 a 是并发的,则在状态 s' 处选择事件 a 重新进行搜索[2].这样,DPOR 只需要将那些不独立的事件进行重排序,而不考虑独立事件之间的重排序,从而约减了状态空间.所以 DPOR 约减的效果取决于独立事件的数量,相互独立的事件越多,DPOR 约减的效果越好.如在下面所示的两个并发进程中,进程 p_1 和进程 p_2 分别对变量 x 和 y 进行赋值.

(1) $p_1:x=1$;

(2) $p_2:y=2$.

如果不使用 DPOR,由于并发进程执行的不确定性,所以这两个进程的执行顺序有两种可能(p_1p_2 和 p_2p_1).然而,我们发现 p_1 和 p_2 两个进程的操作分别对不同的变量进行赋值,交换进程的执行顺序不改变最终到达状态($x=1,y=2$),所以这两个进程对变量的操作符合事件独立性定义,因此使用 DPOR 能够将上面两种可能的执行顺序约减到一条(p_1p_2).

3　本文方法及案例分析

3.1　系统模型

本文提出了一种实现级模型检测方法对分布式系统进行验证,与其他实现级模型检测方法相同[2-5,9,15-16],本方法将待验证分布式系统建模成系统事件交错模型,即将待验证系统建模成一组从初始状态出发的事件执行序列,通过运行系统来获取所有不确定性来源,遍历不确定性来源的所有可能行为从而得到待验证系统的完整模型,即获取到待验证系统所有可能的事件执行序列,执行这些序列从而实现在运行时动态地对分布式系统的全空间进行验证.

本文方法考虑的待验证系统不确定性来源是分布式系统中并发事件执行顺序的不确定性.这些并发事件包括节点间在网络上的通信、节点在本地的读写操作等.因此,本文方法将待验证分布式系统建模成从初始状态出发的一组事件执行序列构成的模型,其中并发事件之间可以进行交错.实现级模型检测在这样的模型上需要搜索并运行每一条事件执行序列来完成验证.但是,那些由于对事件处理存在随机性而引入的不确定选择不在本文的考虑范围内.所以本文方法适用于状态迁移确定的系统,即在给定系统状态 s 下执行一个事件 a,系统只会到达一个确定的唯一的状态 s',如性质 1 所述.

性质 1.　$\forall s(s\xrightarrow{a}s',s\xrightarrow{a}s'')\rightarrow s'=s''$.

3.2　系统规约

如上文所述,分布式系统从初始状态开始存在多条状态迁移序列,这些序列可以用根节点为初始状态的树状结构来表示,可以采用 CTL 逻辑描述分布式系统的规约(后面用性质来表示规约).因为 LTMC 是实现级模型检测器,所以在对目标系统代码进行边运行边验证时,将 CTL 逻辑表述的性质转换成由待验证系统的变量组成的状态逻辑谓词.每个状态逻辑谓词会在给定的状态输出 TRUE 或者 FALSE,如果该状态满足这个谓词,则输出 TRUE,否则输出 FALSE.所以,待验证分布式系统的安全性属性和活性属性采用下面的方法进行表述.

(1)安全性属性.对于系统中的每一个状态都要满足安全性属性,所以系统要满足安全性规约 AGf,其中 f 是安全性属性对应的状态逻辑谓词.因此在待验证系统的每一个状态上,状态逻辑谓词 f 都应该输出 TRUE.

(2)活性属性.对于系统中的每一条状态迁移,总有一个未来的状态满足活性属性,所以系统要满足活性规约 AFg,其中 g 是安全性属性对应的状态逻辑谓词.因此在待验证系统的每一条状态迁移序列上,不需要状态逻辑谓词 g 对所有状态都输出 TRUE,只需在将来存在一个状态 s,状态逻辑谓词 g 在状态 s 上输出 TRUE.

3.3 系统状态空间搜索与属性验证

从上面介绍的安全性属性和活性属性的定义可以发现,对于活性属性的检查与安全性属性完全不同.这是由于对于安全性属性,系统需要在每个状态下都满足该性质;而对于活性属性,系统并不需要在任何状态下都得满足该属性,只需存在未来的某个状态,系统满足该属性.所以,现有的实现级模型检测方法难以检查活性属性,因为需要找到一条无限长度的迁移序列作为反例,其中执行此序列时所遇到的每个状态都不满足活性属性,这样的做法显然是不切实际的.

LTMC 采用第 2 节中介绍的 DFS 作为基础的状态空间搜索方法.对于安全性属性的检查,在执行这些迁移序列的过程中,对每个遇到的系统状态进行检查,检查是否满足系统规定的安全性属性.如果违反了安全性属性,就将对应的这条迁移序列作为反例报告给技术人员,技术人员对其进行分析并对目标系统进行改进.

为了解决上面所述的对于活性属性检查的问题,我们将满足活性属性状态的出现限制在一个有限长度 k 内,从系统初始状态开始执行深度为 k 的 DFS,如果在深度 k 内系统无法满足活性属性则认为该系统可能不满足活性属性,并将这个不满足活性属性的迁移序列报告给技术人员.这样做的原因是:虽然活性可以在无限长的将来被满足,但是在协议设计阶段,这种在无限长时间后才能满足的性质变得毫无意义.例如,在通信传输协议中,其活性属性为一方发送的消息终将被另一发收到,即 AF(receive_message),根据活性属性的定义存在这样一种情况:一方发送的消息在很长的时间之后才被接收,虽然活性属性满足了,但是这样的消息早已失去其时效性,所以这种情况下虽然活性属性被满足,但是失去了现实意义;在分布式系统 Leader Election 协议中,其活性属性规定系统要选举一个 Leader,显然如果系统在开始运行无限长的时间后才满足该性质,那么这个协议就失去其实用性,因为当系统不存在 Leader 的时候是不能够完成其功能的(存储功能等).因此,我们在有限长度内考虑活性属性及其容错性是非常有必要且有意义的.

3.4 故障注入

为了继续对活性属性的容错机制进行验证,将上面提出的系统活性属性检查方法与故障注入技术相结合.在分布式系统中,技术人员认为在理想状态下系统总是能够满足活性属性.为了检查目标系统的活性属性的容错机制,我们定义了关键迁移,如定义 7 所述.在图 2 中,虚线的状态(s_n)满足活性属性,实线的状态(s_{n-1})不满足活性属性,所以从状态 s_{n-1} 到状态 s_n 的迁移 t_{n-1} 为一个关键迁移.

定义 7. 迁移 t 是系统在状态 s 下的一个关键迁移,当且仅当在状态 s 下系统可以执行迁移 t 所对应的事件且 $s \xrightarrow{t} s'$,其中 s' 是系统状态,并且系统状态 s 不满足活性属性而系统状态 s' 满足活性属性.

我们的做法就是在一定成本下(注入故障的数目),每当系统进入一个满足活性属性的状态 s',就将导致状态 s' 出现的关键迁移 t 标记为故障注入点,这样做的目的在于使得关键迁移 t 不能被执行,那么在该时刻下系统由于执行关键迁移 t 而满足的活性属性就会被破坏.之后,从该状态开始重新检查在有限长度 k 内该活性属性是否再次被满足,如果不被满足,说明该分布式系统的活性属性不能够在存在故障的情况下恢复,也就是不能够容错;相反,当活性属性再次满足之后继续进行故障注入操作直至超过规定的最大故障注入数目,若活性属性还能够被满足,那么说明该系统在这个故障数目下的容错机制能够很好地工作.

向分布式系统中注入的具体故障与关键迁移对应操作的类型相关,表 1 列举了一些场景下需要注入的故障类型.例如,如果关键迁移对应节点间的通信,那么可以向通信节点内注入节点崩溃故障或者向通信节点间注入网络故障;如果关键迁移对应节点读写操作,那么可以注入磁盘故障等.

表 1 关键迁移类型与对应注入故障类型

关键迁移对应操作类型	注入故障类型
节点通信	节点崩溃、网络故障
节点内部操作	节点崩溃
读写操作	节点崩溃、磁盘故障

上述方法整体流程如算法 1 所示,首先将待验证系统初始化(第 1~2 行),从初始状态开始调用算法 *Explore* 搜索并检查所有能够执行的事件序列(第 3 行),最后将检查出来的导致 Bug 出现的反例集返回(第 4 行).

算法 1.　基于故障注入的分布式系统协议活性属性容错验证算法.

输入:待验证分布式协议 SUT,搜索深度 k,最大注入故障数 max_I,安全性属性 φ,活性属性 ψ

输出:待验证分布式协议 SUT 的反例集 *CounterExample*

1. *CounterExample*=∅,当前 SUT 状态 S_0,搜索深度 ED=0,已经注入的故障数 $has\_injected$=0;
2. $Current\_State=S_0$;当前迁移序列 $t\_exclusion=\varnothing$;
3. *Explore*(*Current_State*,*t_exclusion*,*has_injected*,*ED*);
4. RETURN *CounterExample*;

算法 2.　*Explore*.

输入:当前搜索状态 *Current_State*,当前迁移序列 *t_exclusion*,已经注入的故障数 *has_injected*,已经搜索的深度 *ED*

输出:待验证分布式协议 SUT 的反例集 *CounterExample*

1. 获取当前 SUT 可以执行的迁移集合 *T_Set*;
2. IF *T_Set*==∅
3. Add(*t_exclusion*,*CounterExample*);
4. ENDIF
5. WHILE *T_Set* !=∅
6. 选择 $t\in T\_Set$; ED++; $T\_Set=T\_Set/t$;
7. 执行 t,得到下一个状态 S;
8. $Next\_State=S$; $t\_exclusion=t\_exclusion+t$;
9. IF S 不满足 φ
10. Add(*t_exclusion*, *CounterExample*);
11. ENDIF
12. IF S 不满足 ψ
13. IF $ED==k$
14. Add(*t_exclusion*, *CounterExample*);
15. CONTINUE;
16. ELSE
17. *Explore*(*Next_State*,*t_exclusion*, *has_injected*,*ED*);
18. ENDIF
19. ELSE
20. IF $has\_injected==max\_I$
21. CONTUINE;
22. ELSE
23. 将系统状态回滚到 *Current_State*;
24. 向系统中注入故障,使得关键迁移 t 被破坏无法执行;
25. $has\_injected$++;
26. *Explore*(*Current_State*,*t_exclusion*, *has_injected*,0);
27. ENDIF
28. ENDIF
29. ENDWHILE

基于 DFS 的故障注入模型检测算法 *Explore* 对当前状态 *Current_State* 进行 DFS 搜索,获取状态 *Current_State* 下可以执行的迁移集(第 1 行),如果在该状态下不存在任何迁移可以执行,那么导致该状态出现的迁移序列则被认为是违反了活性属性(第 2～4 行).每次从可执行迁移集中选择一个迁移并执行,将系统驱动到下一个状态(第 6～8 行),直到超过规定的最大执行深度为止(第 13 行).每当遇到一个新状态时,对其进行安全性属性和活性属性的检查,如果存在不满足安全性属性的状态,就将从初始状态开始导致该状态出现的迁移序列放入反例集中(第 10 行).同时,如果在规定深度 k 内系统还未到达一个满足活性属性的状态,则将导致其出现的迁移序列放入反例集中(第 14 行).如果在搜索过程中遇到一个满足活性属性的状态,将系统回滚到上一个状态,并在系统中注入故障来破坏关键迁移 t,重新搜索 k 步,检查系统是否能够在 k 步内重新满足活性属性(第 23～26 行),重复上面的过程直到系统已经注入故障数等于最大故障数目(第 20 行).当在初始状态下没有可执行事件时,算法 *Explore* 停止(第 5 行).为了将反例集中的迁移序列与安全性属性 Bug 和活性属性 Bug 对应,可以在将其加入反例集之前打上分类标签.

3.5　状态空间约减

在 3.4 节介绍的算法中,我们采用的是深度优先的状态搜索策略,但是随着最大深度 k 的变大,算法搜索出的事件序列数量呈指数增长.为了使得模型检测能够具有实用性,所以需要一些策略来对需要搜索的状态空间进行约减,即约减需要搜索的事件序列数量.

为了对事件序列数进行约减,算法 3 在 DFS 的基础上结合第 2.2.3 节介绍的 DPOR[2,17] 思想来约减需要搜索的事件序列数量,并对搜索出的状态进行检查.在进行模型检测时,将算法 1 中的使用的 DFS 搜索算法 *Explore* 更改为算法 3 所示的 DPOR 搜索,在算法 3 中,我们使用了如下几个标记:

(1) $dom(i)$ 表示集合 $\{0,1,\cdots,i\}$.

(2) $t\_exclusion_i$ 表示在当前执行路径上的第 i 个迁移.

(3) S_i 表示在当前执行路径上执行第 i 个迁移的状态.

算法 3.　基于 DPOR 与故障注入的分布式系统协议活性属性容错验证算法 *DPORExplore*.

输入:当前搜索状态 *Current_State*,当前迁移序列 *t_exclusion*,已经注入的故障数 *has_injected*,已

```
        经搜索的深度 ED
输出:待验证分布式协议 SUT 的反例集 CounterExample
1.  获取当前 SUT 可以执行的迁移集合 T_Set;
2.  IF T_Set==∅
3.    Add(t_exclusion,CounterExample);
4.  ENDIF
5.  IF ∃t∈T_Se T
6.    Backtrack(Current_State)={t};Done={∅};
7.    While(∃t'∈Backtrack(Current_State)/Done)
8.      Add(t',Done);
9.      IF(∃i=max(i∈dom(ED)|t'与t_exclusion_i不
        独立且 t'∈enable(S_i)))
10.       Add(t',Backtrack(S_i));
11.     ENDIF
12.     执行 t',得到下一个状态 S;ED++;
13.     t_exclusion=t_exclusion+t;
14.     IF S 不满足 φ
15.       Add(t_exclusion,CounterExample);
16.     ENDIF
17.     IF S 不满足 ψ
18.       IF ED==k
19.         Add(t_exclusion,CounterExample);
20.         CONTINUE;
21.       ELSE
22.         DPOR(S,t_exclusion,has_injected,ED);
23.       ENDIF
24.     ELSE
25.       IF has_injected ==max_I
26.         CONTUINE;
27.       ELSE
28.         将系统状态回滚到 Current_State;
29.         向系统中注入故障,使得关键迁移 t,被破
            坏无法执行;
30.         has_injected++;
31.         DPOR(Current_State,t_exclusion,
                 has_injected,0);
32.       ENDIF
33.     ENDIF
34.   ENDWHILE
35. ENDIF
```

算法 3 利用迁移之间的非独立关系,在状态搜索过程中动态地找出当前事件执行路径上的回溯搜索点 S_i,其中 S_i 满足:$t\_exclusion_i$ 为离当前迁移 t' 最近的非独立的已执行迁移且 t' 在状态 S_i 下可以执行(第 9 行),那么将 t' 添加进 S_i 的回溯集中作为回溯执行的迁移(第 10 行).同时与算法 2 类似,对每个搜索出的状态进行安全性属性和活性属性检测(第 12～35 行).

同时,由于在分布式系统中存在大量的冗余节点和对等节点,这些节点在系统中的角色和功能是相同的,对其中任何一个节点的相同操作都是等价的,因此,由对不同的对等节点进行相同的操作的先后顺序不同而产生的序列是等价的,可以进行约减,将这种策略称为对等约减 PRP(Peer Reduction Policy).定义 8 给出对等节点的定义,定义 9 对两条迁移对应操作是否相同给出定义.

定义 8. 在一时刻下,节点 N_1 和 N_2 是对等节点,当且仅当下面两个条件同时满足,其中 set_{fN_i} 表示第 i 个节点在该时刻下可执行的功能集合:

(1) $\forall f1_i \in set_{fN_1}, \exists f2_k \in set_{fN_2}, f2_k = f1_i$.

(2) $\forall f2_i \in set_{fN_2}, \exists f1_k \in set_{fN_1}, f1_k = f2_i$.

从定义 8 可以看出,在分布式系统中,所有功能相同的节点都是对等节点.例如在分布式存储系统中,所有的副本节点即为对等节点,在去中心化的比特币网络中,所有的轻节点为对等节点.

定义 9. 迁移 t_1 与 t_2 对应的操作内容相同当且仅当下面两个条件同时满足:

(1) t_1 与 t_2 操作的对象相同或者为不同节点中一致的对象.

(2) t_1 与 t_2 对操作对象执行的行为相同.

从定义 9 可以看出,例如,在分布式系统中迁移 t_1 将消息 M 从节点 1 传递给节点 2,迁移 t_2 对应的操作将消息 M 从节点 3 传递给节点 4,由于迁移 t_1 与 t_2 操作的对象都为消息 M 且对 M 的行为都为传递消息,所以迁移 t_1 与 t_2 对应的操作内容相同;同样地,如果迁移 t_1 将节点 1 中的变量 a 进行“+1”操作,迁移 t_2 将节点 2 中的变量 a 进行“+1”操作,由于 t_1 与 t_2 的操作的对象为不同节点中一致的对象 a,且行为相同(都为“+1”操作),所以迁移 t_1 与 t_2 对应的操作内容相同.在如图 3 所示的简易比特币网络中,这是一个 P2P 的分布式网络系统,当一个矿工节点计算出一个区块时,需要将该区块进行全网广播,其他节点收到该区块并对其进行验证,验证通过后将该区块加到区块链的尾端.在比特币的网络协议中,广播操作规定,节点只将区块发送给一个邻居节点,当区块验证通过后再由当前节点发送给另外一个新的邻居节点,依次类推,从而实现全网的广播.假设节点 A 计算出一个新的区块 Block1,需要将 Block1 进行全网的广播,在不考虑拓扑距离的情况下,Block1 在该网络上的传播会出现下面 6 种情况:

1. *A*-*B*, *B*-*C*, *C*-*D*
2. *A*-*B*, *B*-*D*, *D*-*C*
3. *A*-*C*, *C*-*B*, *B*-*D*
4. *A*-*C*, *C*-*D*, *D*-*B*
5. *A*-*D*, *D*-*C*, *C*-*B*
6. *A*-*D*, *D*-*B*, *B*-*C*

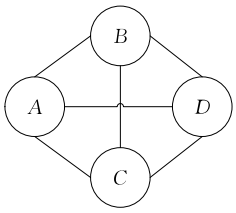


图 3　4 个节点的简易比特币网络示意图

在该系统中节点 *B*、*C*、*D* 具有相同的功能(广播区块、验证区块和存储区块),所以这三个节点互为对等节点,且上面这 6 条执行序列是由于节点 *A* 对这三个对等节点进行相同的操作(发送 Block1)产生的,因此使用 PRP 策略,上述 6 条序列可以被约减到 2 条:*A*-*B*,*B*-*C*,*C*-*D* 和 *A*-*B*,*B*-*D*,*D*-*C*. 可以发现这两条序列也是由于节点 *B* 对对等节点 *C* 和 *D* 执行的相同操作产生的,所以继续使用 PRP 策略,最终只需要对上面 6 条序列中的 1 条序列(*A*-*B*,*B*-*C*,*C*-*D*)进行搜索和检测即可,大大约减了需要搜索的序列数量和状态空间.

为保证 PRP 状态约减不会影响性质的验证,该策略只对执行的先后顺序与待验证性质无关的迁移进行约减,用 *independent*(*t*1,*t*2)表示迁移 *t*1 和 *t*2 的先后关系与待验证性质无关. 例如,在一个系统中有两个对等节点 *A* 和 *B*,待验证性质为节点 *A* 收到消息后节点 *B* 再收到消息. 在这种情况下,虽然节点 *A* 收到消息与节点 *B* 收到消息这两条迁移的目标节点为对等节点且迁移的操作内容相同,但是由于这两条迁移的先后顺序与待验证性质相关,所以不能进行约减.

对等约减策略 PRP 的思路如算法 4 所示. 每当算法 *Explore* 在获取到当前可以执行的迁移集合 *T_Set* 时(算法 2 第 1 行)调用算法 4,来约减需要进行搜索的操作数量. 下面对系统状态迁移的目标节点给出定义.

定义 10.　对于一个迁移 *t*,其目标节点分两种情况考虑:

(1) 如果 *t* 对应的操作是节点内部操作,目标节点为该操作发生的节点.

(2) 如果 *t* 对应的操作是节点间的操作,例如节点 *A* 向节点 *B* 发起的一次操作,那么目标节点即为节点 *B*.

算法 4.　对等约减策略算法 *PRP*.

输入:待验证分布式协议 SUT,当前状态下可以执行的迁移集合 *T_Set*

输出:约减后的 *T_Set*

```
1.  FOR t1∈T_Set
2.    FOR t2∈T_Set
3.      IF ispeer(t1.target,t2.target)&&t1.content==
          t2.content&&independent(t1,t2)
4.        T_Set/t2;
5.      ENDIF
6.    ENDFOR
7.  ENDFOR
8.  RETURN T_Set;
```

算法 4 从当前可以执行的迁移集合 *T_Set* 中任取两个迁移(第 1～2 行),如果这两个状态迁移的目标节点在系统中是对等节点、迁移对应的操作内容相同且迁移的先后顺序与待验证性质无关(第 3 行),那么这两个迁移就是对等冗余的,将其中一个从需要搜索的迁移集合中去除(第 4 行),直到遍历完该集合中的所有迁移.

从在上面的例子中可以发现,这 6 条执行序列中不存在独立事件,所以使用 DPOR 策略无法对其进行约减,所以本文提出的对等约减和 DPOR 方法是互补的,可以在 DPOR 的基础上,对状态空间继续进行约减.

3.6　增量状态搜索

可以发现在 3.3 节提出的 DFS 方法存在状态搜索深度设置过小的问题,在这种情况下,待验证系统未得到充分的机会来满足活性属性,所以得到的违反活性属性的反例可能存在假阳性. 同时,在真实世界的系统中,状态空间中的迁移数量往往十分巨大或者系统不存在终止状态,那么在这些状态空间下,实现级模型检测想要执行所有的迁移来遍历状态空间是不实际的,为了在遍历状态空间和搜索可行性之间取得平衡,我们对违反活性属性的反例序列采用增量式状态搜索,对于反例序列每次增加搜索深度 1 步,继续进行搜索. 这样既能够在一定程度上提供给未满足活性属性的序列在未来满足活性属性的可能性,同时可以根据验证成本动态地结束搜索过程.

3.7　时钟向量

从 2.2 节介绍的两种公平性原则可以看出,一个被使能的进程不希望被长时间忽略而未得到执行. 根据这个思想,我们使用时钟向量 ***CV*** 来保证不会有被使能的节点长时间地不执行. 时钟向量 ***CV*** 是一个节点标识符到整数映射的 *map*.

$$\boldsymbol{CV}=\mathcal{P}\rightarrow\mathcal{N}.$$

对于有 n 个节点的分布式系统，$\boldsymbol{CV}(\mathcal{P}_i)$ 表示在当前状态与第 i 个节点最近一次未被执行的使能迁移的初始使能状态的距离. $\boldsymbol{CV}(\mathcal{P}_i)$ 越大表明第 i 个节点等待被执行的时间越长；当 $\boldsymbol{CV}(\mathcal{P}_i)=0$ 时，则表明当前状态下执行了第 i 个节点的迁移；当 $\boldsymbol{CV}(\mathcal{P}_i)<0$ 时，则表明不存在第 i 个节点的迁移被使能. 下面给出时钟向量的运算：$\boldsymbol{CV}[\mathcal{P}_i=c_i']=\langle c_1,\cdots,c_{i-1},c_i',c_{i+1},\cdots,c_n\rangle$. 定义 11 给出迁移 t 的执行节点的定义，并用 $Proc(t)$ 来表示执行迁移 t 的节点.

定义 11. 一个迁移 t 被节点 A 执行当且仅当：

(1) t 对应的操作是节点 A 的内部操作或者

(2) t 对应的操作是节点间的操作，且该操作为节点 A 向另一节点 B 发起的.

为了使得被使能的节点不会长时间地不被执行，我们在搜索状态时优先选择最久未被执行的节点执行相应的迁移，保证先搜索出来的执行路径不会违反公平性原则，从而使得先暴露出的 Bug 更具有现实意义.

如算法 5 所示，在状态搜索开始前，需要将时钟向量 $\boldsymbol{CV}$ 初始化，将每个分量初始化为 -1. 在执行状态搜索时，以算法 2 为例，我们对其进行改进，其他状态搜索算法类似. 当获取当前可执行迁移集合 T_Set 时，将新被使能的节点的时钟分量置为 0(第 5～9 行)；当选择一个迁移 t 并进行后续的状态搜索时，需要更新时钟向量 $\boldsymbol{CV}$，将迁移 t 对应节点的时钟分量置为 -1，其余被使能节点的时钟分量加 1(第 14 行).

算法 5. *ExploreWithCV.*

输入：当前搜索状态 *Current_State*，当前迁移序列 *t_exclusion*，已经注入的故障数 *has_injected*，已经搜索的深度 *ED*，时钟向量 $\boldsymbol{CV}$

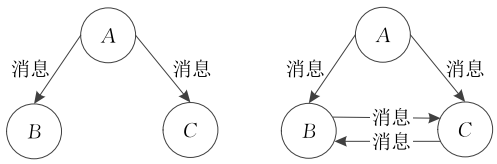
输出：待验证分布式协议 SUT 的反例集 *CounterExample*

```
1.  获取当前 SUT 可以执行的迁移集合 T_Set;
2.  IF T_Set==∅
3.     Add(t_exclusion,CounterExample);
4.  ENDIF
5.  FOR (t∈T_Set)
6.     IF CV(Proc(t))<0
7.        CV(Proc(t))=0;
8.     ENDIF
9.  ENDFOR
10. WHILE T_Set !=∅
11.    选择 t∈T_Set 且 CV(Proc(t))最大;
12.    ED++; T_Set=T_Set/t;
13.    执行 t,得到下一个状态 S;
14.    CV[Proc(t)=-1][Proc({t'|CV(Proc(t'))>
                    0})++];
15.    Next_State=S; t_exclusion=t_exclusion+t;
16.    IF S 不满足 φ
17.       Add(t_exclusion,CounterExample);
18.    ENDIF
19.    IF S 不满足 ψ
20.       IF ED==k
21.          Add(t_exclusion,CounterExample);
22.          CONTINUE;
23.       ELSE
24.          Explore(Next_State,t_exclusion,has_injected,
                    ED,CV);
25.       ENDIF
26.    ELSE
27.       IF has_injected ==max_I
28.          CONTUINE;
29.       ELSE
30.          将系统状态回滚到 Current_State;
31.          向系统中注入故障,使得关键迁移 t 被破坏
             无法执行;
32.          has_injected++;
33.          Explore(Current_State,t_exclusion,has_injected,
                    0,CV);
34.       ENDIF
35.    ENDIF
36. ENDWHILE
```

使用 PRP 策略进行约减时(算法 4)，如果两个迁移 t_1、t_2 符合对等约减，保留对应时钟分量大的迁移，这样可以将长时间没有执行的迁移先执行，其余迁移留在在未来执行.

3.8 案例分析

在本节中，我们将通过在分布式系统中普遍使用的广播协议来对本文提出的方法进行具体案例分析. 图 4(a)是一个有三个节点的分布式系统，其中节点 A 是领导者，节点 B 和 C 是跟随者. 根据简单原子广播协议，节点 A 要分别向节点 B 和节点 C 发送一条相同的消息. 在上述的协议中，安全性属性是：系统不会出现死锁、内存溢出等情况；活性属性是：在未来的某一时刻，系统中所有信道中不会存在



(a) 简单原子广播协议 (b) 带转发的原子广播协议

图 4 案例分析示意图

正在发送的消息并且每一个正在运行的节点都会收到被广播的消息.

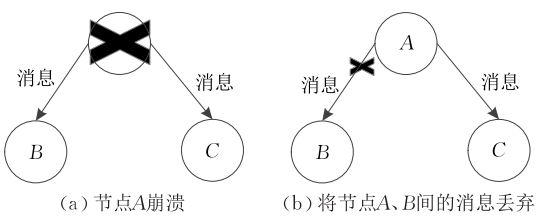
在图 4(a)所述的协议上使用我们所提出的方法,设最大故障注入数为 1,搜索深度为 3.若不注入故障,那么搜索出 2 条可能的执行序列;当注入 1 个故障时,那么搜索出来 4 条可能的执行序列,如表 2 所示.表 2 展示了分别在不注入故障和注入故障情况下,使用算法 1 搜索出来的事件执行序列,其中加粗标出的是向系统中注入的故障(*X-Y* 代表节点 *Y* 收到 *X* 节点发送的消息,*CrashX* 代表节点 *X* 崩溃,*OmitXY* 代表当前节点 *XY* 之间的信道发生故障无法传递消息).

表 2　案例分析结果

序号	不进行故障注入	进行故障注入
1	*A-C*, *A-B*	*A-C*, ***CrashA***
		A-C, ***OmitAB***
2	*A-B*, *A-C*	*A-B*, ***CrashA***
		A-B, ***OmitAC***

我们可以看到在图 4(a)所示的协议中存在两种可能的关键迁移,在第一种情况下节点 *C* 先收到节点 *A* 的消息,然后节点 *B* 再收到节点 *A* 的消息,此时系统的活性属性被满足,那么 *A-B* 就是关键迁移对应的事件.由于 *A-C* 和 *A-B* 之间发生先后的不确定性,所以在第二种情况下,节点 *B* 先收到节点 *A* 的消息,然后节点 *C* 再收到节点 *A* 的消息,同理 *A-C* 就是关键迁移对应的事件.

根据我们的算法,我们需要破坏关键迁移,由于迁移都是因为节点收到并处理其他节点发送的消息而产生的,所以我们只需要让节点在当前时刻收不到关键迁移对应的消息.具体地,对于 *A-B* 而言,可以让节点 *A* 发生崩溃(图 5(a)所示),这样节点 *A* 就不会发送消息给节点 *B*;同时,还可以向节点 *A* 与节点 *B* 之间的信道中注入网络故障,使节点 *B* 无法收到节点 *A* 发送的消息即 *OmitAB*(图 5(b)所示),这两种情况下,系统都不存在可执行的迁移,所以节点 *B* 无法收到节点 *A* 的广播消息,这就是一个违反活性属性的 Bug;同理,对于 *A-C* 而言,可以注入 *CrashA* 和 *OmitAC* 两种故障来破坏关键迁移 *A-C*.所以,使用算法 2 可以得出图 4(a)的协议在注入 1 个故障的情况下不具备容错性,并且能够追踪到导致该 Bug 出现的 4 条事件执行序列(如表 2 第 3 列所示).



(a) 节点*A*崩溃　　(b) 将节点*A*、*B*间的消息丢弃

图 5　破坏关键迁移示意图

由于事件 *A-C* 和 *A-B* 可以交换执行顺序,并且无论以何种顺序执行系统都到达相同的状态(节点 *B*、*C* 都收到节点 *A* 发出的消息),所以事件 *A-C* 和 *A-B* 相互独立.对该例子使用 DPOR 方法,当系统在搜索执行表 2 中的第 1 条序列的事件 *A-B* 时,不存在与之不独立的事件,所以对于该例子而言,只需搜索第 1 条事件序列并对其进行故障注入即可,而表 2 中第 2 条执行序列(*A-B*,*A-C*)以及相应的故障注入序列就被约减了.

当使用对等约减策略 PRP 时,因为节点 *B*、*C* 的角色相同且节点 *A* 对节点 *B* 和 *C* 的操作是相同的,所以对等约减策略同样可以将上面 2 条执行序列约减到 1 条,只需要搜索第一条序列并进行故障注入即可.

4　LTMC 实现以及实验分析

LTMC 结构如图 6 所示,在分布式系统的每一个节点上引入一个控制层(图 6 中节点灰色部分),控制层获取并阻塞该节点当前可以执行的操作,在后端引入一个能够与每个节点控制层进行 RMI 通信的服务器.这样,每个时刻控制层都会将该时刻节点能够发生的事件(即并发事件)进行阻塞并发送给服务器,服务器收到这些事件之后使用 DFS 的状态空间搜索方法对其进行重排序,服务器根据重排之后的事件执行顺序将阻塞的事件进行使能(图 6 中的执行 *a*).这样待验证系统就能够按照特定顺序执行特定的事件,得到下一时刻的系统状态,并在该状态下检查安全性属性和活性属性,之后在得到的状态上再次进行上述操作,从而实现系统状态空间的

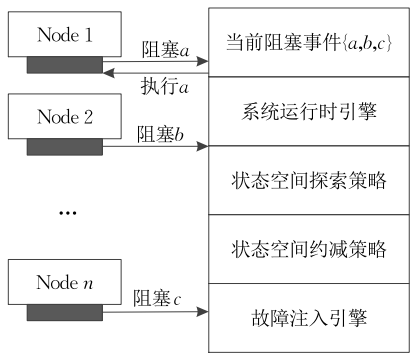


图 6　LTMC 结构模型

遍历.服务器可以使用对等约减策略 PRP 和 DPOR 来减少需要执行的事件序列数.同时服务器能够主动地使用本文提出的方法检测到故障注入点,并向待验证系统中注入故障以验证分布式系统中活性属性的容错性.

我们使用 Java AspectJ 技术实现了 LTMC 的控制层.AspectJ 是一种面向切面的编程技术,允许在不改变目标系统源代码的基础上,获取感兴趣的待验证系统的状态和变量.当待验证的系统发生变化时,我们只需要对控制层进行一些简单的修改,这样 LTMC 就可以快速地对使用 Java 编写的不同协议进行模型检测.

我们采用虚拟机环境(VM)来实现 LTMC,每台虚拟机相互独立,各自运行待验证系统的代码,从初始状态开始加载工作负载,进行状态搜索.采用虚拟机环境有如下优势:首先,虚拟机环境可以记录待验证系统的执行路径,以便对出现的不可预料的行为进行研究;第二,随着云技术的逐渐成熟,虚拟机能够充分利用异构设备的计算能力.然而,采用虚拟机也会带来一些不可避免的额外时间开销.当恢复待验证系统时,需要花费几秒钟的时间,该开销限制了在给定的时间预算内可以搜索到的待验证系统执行路径数.

4.1 实验问题

LTMC 的目的是有效地发现分布式协议中安全性属性、活性属性及其容错机制中存在的 Bug.为了验证 LTMC 发现 Bug 的能力,我们提出实验问题 1;为了验证 LTMC 报告 Bug 的真实性,我们提出实验问题 2;为了验证 LTMC 中使用约减策略的有效性,我们提出实验问题 3.为了验证 LTMC 的可拓展性,我们提出了实验问题 4.

(1)实验问题 1. LTMC 能否在指定验证成本下发现分布式协议中存在的 Bug?

(2)实验问题 2. LTMC 报告的 Bug 是否存在误报?

(3)实验问题 3. LTMC 使用 DPOR 与对等约减 PRP 之后,是否减少了搜索序列的数目?

(4)实验问题 4. LTMC 的可拓展性如何?

4.2 实验对象

为了回答提出的实验问题,我们选取 Zookeeper 和 Cassandra 这两个广泛使用的分布式系统中的三个协议作为实验对象,其中 Zooleeper 选用 3.4.14 版本,Cassandra 选用 1.0.0 版本.我们对 Zookeeper 的领导者选择协议(ZLE)、原子广播协议(ZAB)以及 Cassandra v1.0.0 版本的 Gossiper 协议(GS)进行实验.

ZooKeeper 是一个分布式服务框架,用来解决分布式应用中的一些数据管理问题.Zookeeper 需要保证全局数据一致,集群中每个服务器保存一份相同的数据副本,客户端无论连接到哪个服务器,展示的数据都是一致的.为了实现这个特性,Zookeeper 使用领导者选择协议在服务器中选择一个 Leader 作为事务请求的唯一调度和处理者,使用原子广播协议保证其余服务器与 Leader 数据的一致性.ZLE 解决了如何从所有服务器中选举一个 Leader,所以其活性属性是系统存在一个 Leader;ZAB 解决了对于某个写操作如何保证所有服务器上数据全被一致更新,所以其活性属性是对于某个写操作,所有节点维持数据一致.

Cassandra 是分布式 Key-Value 存储系统,Cassandra 集群使用 Gossiper 协议,定期交换节点的状态信息,从而实现节点状态的一致,所以 GS 的活性属性为系统所有节点状态保持一致.

4.3 实验结果分析

我们使用 LTMC 对 Zooleeper v3.4.14 的领导者选择协议(ZLE)、原子广播协议(ZAB)以及 Cassandra v1.0.0 的 Gossiper(GS)协议进行实验,实验结果如表 3 所示,其中第 1 列指出目标协议的

表 3 LTMC 实验结果

协议	搜索方法	搜索深度 k	注入故障	执行序列数	加速	Bugs	误报
ZAB	DFS	20	1	66		L	0
ZAB	PRP	20	1	13	5.1	L	0
ZAB	DPOR	20	1	18	3.7	L	0
ZLE	DFS	20	1	3542^{+}		L	0
ZLE	PRP	20	1	264	13.4	L	0
ZLE	DPOR	20	1	221	16.0	L	0
GS	DFS	20	1	580		NO	*
GS	PRP	20	1	3	193.3	NO	*
GS	DPOR	20	1	40	14.5	NO	*
GS	DFS	20	2	4161^{+}		L	0
GS	PRP	20	2	7	594.4	L	0
GS	DPOR	20	2	101	41.2	L	0

名称;第 2 列指出使用的搜索方法;第 3 列给出状态搜索深度 k;第 4 列给出注入故障数;第 5 列给出 LTMC 搜索的执行序列数目($^+$表示超出 12 h 执行时间);第 6 列给出 PRP 和 DPOR 方法相对于 DFS 的加速倍数;第 7 列 Bugs 栏指出发现 Bug 的类型(L 是指在发现违反活性属性的反例).第 8 列指出误报反例的数目(*表示未发现反例).

对于实验问题 1,从表 3 中可以发现:使用 DFS、DPOR 和 PRP 策略都可以有效地发现这些待验证协议中存在的 Bug.而对 GS 协议在注入故障数为 1 的情况下,活性属性能够恢复且不存在违反活性属性和安全性属性的反例.

对于实验问题 2,从表 3 中的误报列可以看出,LTMC 报告的 Bug 都是真实存在的 Bug,不会出现误报.这是因为 LTMC 是实现级模型检测工具,LTMC 对目标协议进行边运行边验证,因此所有出现的 Bug 都是由真实可发生的分布式系统事件执行序列所导致的.

对于实验问题 3,LTMC 在使用 DPOR 与 PRP 方法后执行序列数目相比 DFS 显著减少,至少可以达到 3.7 倍的约减效果.同时只要是 DFS 方法能够发现的 Bug,DPOR 和 PRP 方法都可以使用更少的执行序列来发现同样的 Bug.并且,我们可以发现当涉及的对等节点操作越多,PRP 方法比 DPOR 方法的效果越好,例如在 GS 协议中,由于 GS 是 P2P 网络协议,所有节点都是对等的,所以使用 PRP 的效果是 DPOR 的 13.3 倍以上.然而在一些独立性事件多,对等操作少的场景中,DPOR 约减效果更好,例如在 ZLE 协议中,由于节点角色发生改变,对等操作数量较少,所以,使用 DPOR 的效果比 PRP 效果好.因此 DPOR 与 PRP 方法是相辅相成、相互补充的.

对于实验问题 4,我们增加实验对象的规模,在不同规模下对比 DFS 和 PRP 两种方法在不注入故障、搜索深度 k 为 20 时遍历状态空间所需要搜索的执行路径数,实验结果如图 7 所示.从图 7 可以看出在不同节点规模下,PRP 方法搜索的执行路径远远低于 DFS 方法.随着节点数目的增加,待验证系统中的并发事件随之增多,DFS 方法在节点数为 4(ZLE)、5(ZAB 和 GS)时,搜索时间已经超过 12h,搜索出的执行路径序列超过 5000,而 PRP 能够在 10 个节点的规模下在可接受时间内完成搜索.从图 7 中可以看出,PRP 方法在 GS 协议上效果最好,在 10 个节点的规模下,只需搜索一条执行序列即可等价遍历整个空间.为了研究故障注入数对 LTMC 的影响,我们对 10 个节点的 GS 协议进行了实验,每次增加一个故障,得到表 4 的结果.从表 4 可以看出,在相同搜索深度 20 下,随着注入故障数的增多,搜索的执行路径也变多,同时能够暴露 Bug 的路径数也增多.同时,如果注入故障过少,无法将系统中的 Bug 暴露出来,所以在使用 LTMC 时,可以先注入少量的故障,如果时间成本允许,再注入更多的故障,更好地将系统中的 Bug 暴露出来.

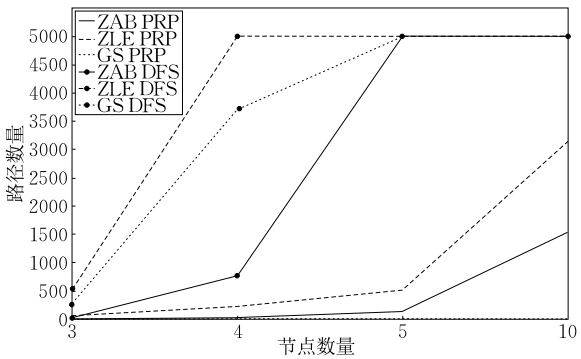


图 7　LTMC 在不同系统规模下的实验结果

表 4　注入故障数对 LTMC 的影响

协议	搜索深度 k	故障注入数	执行路径数	暴露 Bug 路径数
GS	20	0	1	0
GS	20	1	3	0
GS	20	2	7	1
GS	20	3	11	3
GS	20	4	15	6

5　相关工作

在分布式系统可靠性与容错性研究领域,研究人员从多种角度提出了不同的研究方法.模型检测技术和故障注入测试是其重要的组成部分.

5.1　模型检测技术

近年来,模型检测技术在验证软件系统的正确性方面取得较大的进展.如今,随着分布式系统不断发展,分布式系统中由于不确定性带来的安全问题愈发严重,因此验证分布式系统的正确性成为一个巨大的挑战.目前,已有很多研究使用模型检测技术来对分布式系统进行检验.在分布式系统中运用模型检测技术是对整个目标系统进行建模,通过遍历其状态空间对其进行验证.目前有两种常用的建模方法:

(1) 将分布式系统建模成抽象的可计算轨迹——状态机[10-12]、Petri 网[13-14]等来建模,称为抽象模型检测[19].

(2)直接将系统的源码建模为运行时模型.具体而言,由于目标系统的每一个状态以及状态迁移都是由系统执行了某些事件而导致的,并且由于并发性,目标系统有多种可能的事件执行序列,所以第二种方法通过控制系统的运行,执行所有可能的事件执行序列,从而遍历整个状态空间,称为实现级模型检测[16].

5.1.1 抽象模型检测

抽象模型检测方法,使用时序逻辑(CTL[20-21]、LTL[20]、CTL*[22-24]等),根据系统设计,对系统进行抽象建模,一般将系统建模为状态迁移的模型——状态机[10-12]、Petri 网[13-14]等.对于 CTL,技术人员开发了 SMV[25],Gammie 等人[26]开发了 MCK,Lomuscio 等人[10-11,27]开发了 MCMAS,这些工具能够检查给定系统模型与 CTL 规格是否一致;对于实时的时序逻辑 TCTL,Wang 在 1997 年开发了 UPPAAL[28];对于线性时序逻辑 PLTL,SPIN[25]是运用最广泛工具之一.虽然这些工具都有近二十年的历史,但目前研究人员还在对这些工具进行改进,使它们能够高效地应对目前的分布式场景.同时,近年来研究人员发现,使用上面介绍的模型检测工具可以对目标系统系统进行仿真,能够对模型的一些性质进行定量的测量,再利用统计学方法根据模型的定量测量结果对目标系统进行分析.Nigro 和 Sciammarella[29-31]使用 UPPAAL 工具对分布式实时系统进行了仿真,根据多次仿真得到的数据对模型进行统计学分析验证.Jensen 等人[13-14]利用 Petri 网,通过库所表示资源并对库所加上时间限制来对系统以及时间进行建模,同时提出了 Petri 网验证工具 CPN 工具.然而,上面介绍的方法都是基于待验证系统的抽象模型进行验证的,只能验证系统模型与规格是否一致,而在系统实现过程中由于代码编写所引入的 Bug 却无法被找出.

5.1.2 实现级模型检测

实现级模型检测对已经使用高级语言实现的分布式系统代码进行运行时验证,通过控制待验证系统的运行将其驱动到所有可能的状态.所以,该方法需要考虑目标系统的每一条真实的事件执行序列.但是随着系统中并发事件数量的增加,系统执行的事件序列数目呈指数增长,这种巨大的执行序列数目在实际应用中是无法接受的,所以必须对执行序列数量进行约减.下面介绍近年来科研人员对执行序列数量约减的研究.

Guerraoui 等人[32]提出在分布式系统中每个节点都是可以独立工作的,而整个系统的状态是由每个节点的局部状态组合而成的,如果能够保证每个节点都能正确运行,那么在一定程度上就能保证整个系统能够正确无误地运行.由于该方法没有考虑到将所有节点相结合而组成的全局状态,所以舍弃了一部分与全局状态有关的状态空间,从而实现了状态空间的约减,但是这样做却不能完全发现系统中所有 Bug,因为有些 Bug 是和全局状态有关的.

基于 DPOR 方法,Simsa 等人[33]提出了模型检测工具 dBug,通过黑盒分析方法,细化了对事件独立性的定义,使用细化的 DPOR 方法对状态空间进行约减.Leesatapornwongsa 等人[15-16]开发出 SAMC 工具,通过白盒分析的方法,根据目标系统源代码中系统对分布式事件(处理网络消息、节点崩溃等)的处理逻辑的语义分析,可以更加细致地判断两个事件之间是否相互独立,增加系统中独立事件的数量,由于 DPOR 方法通过忽略独立事件间的重排序实现空间约减,能够在传统 DPOR 的基础上对状态空间进行二次约减,所以使用该方法能够再次约减状态空间,减少需要执行的事件序列的数量.

上面介绍的模型检测方法都是用来检查目标系统中是否存在违反某些规格的 Bug.此外,Yabandeh 等人[34]实现了一种特殊的模型检测工具 Crystallball,Crystallball 能够从当前系统状态开始预测未来一段时间内系统所有可能的状态,在预测时,如果发现系统可能会遇到 Bug,则自动地找出导致该 Bug 出现的事件,并阻止该事件发生,从而让系统偏离导致 Bug 出现的路径,使得系统能够保持正确地运行.这种模型检测方法与其他的方法的不同之处在于,一般的模型检测目的在于将目标系统驱动到一个错误的状态以找出导致 Bug 的原因,让技术人员能够进行修复;而 Crystallball 则不同,在发现 Bug 的同时能够阻止 Bug 的发生,自动地将目标系统驱动到正确的运行轨迹上去.

5.2 故障注入测试

故障注入作为对分布式系统容错机制的测试方法之一,大量的研究工作指出不同的研究思路并提出相应的测试方法和工具.Gunawi 等人提出了一种在线故障预演的新型云服务 FaaS[35],其认为线下测试有很多故障场景难以被覆盖,因此提出了故障预演,与其等待故障发生,不如预演故障发生的场景;同时,Gunawi 等人还提出了用于测试分布式系统故障恢复机制的测试框架 FATE 和 DESTINI[36],并

在此基础上进行改进,开发出 PREFAIL[37],可以支持经验丰富的技术人员编写自己的故障空间约减策略,对多个故障的组合空间进行剪枝,并进行故障注入. Netflix 工程师在随机测试的基础上,基于 Chaos Engineering 思想提出了 Chaos Monkey[38],这是一种基于随机注入的思想在分布式系统中进行故障注入的测试方法. Joshi 等人[35]提出了一种基于扰乱系统的大规模分布式系统测试框架 SETSUDO[39],通过向系统中注入故障来扰乱待测系统,分析系统内部状态,从而暴露出系统中的 Bug.

目前对分布式系统可靠性的研究工作十分丰富,也有一些工作将模型检测与故障注入技术相结合,例如 dBug 和 SAMC. 这两个工具都支持故障注入,但其注入故障的方式是穷举注入,缺乏目的性,对所有时间点都进行故障注入,因此会造成由于注入故障而引入的验证状态空间爆炸的问题. 为了解决该问题,这些工具限制了能够注入的故障总量,从而避免了状态空间爆炸. 本文提出了一个关键迁移的概念以及一种有目标的故障注入方法,将原先故障的穷举注入限制到在关键迁移处注入故障,这样既约减了由于注入故障而引入的状态空间,并且增加了可以向系统内注入的故障总数.

6　总结与展望

目前大量使用分布式系统来为软件提供数据存储和处理等服务,使得应用程序变得越来越轻薄,但是由于分布式系统的复杂性、并发性以及节点的不可靠性,保证分布式系统的正确性和容错性是十分困难的,单纯通过软件测试的方法只能说明在测试用例代表的这些情况中不会出现 Bug,无法证明在任何情况下 Bug 都不会出现,因此保证分布式系统的正确性和容错性是目前的挑战.

使用形式化方法可以很好地验证分布式系统的正确性和容错性,其中模型检测是一种使用广泛的且高效的形式化方法. 本文提出了一种将故障注入与模型检测相结合的方法并完成了一个实现级模型检测工具 LTMC 来验证分布式系统协议的安全性属性、活性属性的正确性,以及活性属性的容错性. LTMC 使用一种新的有目标的故障注入选择点方法,避免了随机故障注入的偶然性,同时也在一定程度上约减了穷举故障注入带来的状态空间. 同时,本文提出了一种利用分布式系统节点角色对等关系的对等约减策略 PRP,能够有效地减少需要搜索的状态空间. 通过实验,证明了 LTMC 能够在一定成本下有效地发现待验证系统中所有违反安全性属性和活性属性的 Bug 以及活性属性容错机制中的 Bug.

待验证系统的状态空间爆炸问题往往限制了模型检测技术的应用,在未来的研究工作中,我们将继续研究如何通过待验证系统的代码内部逻辑,自动获取可以进行约减的条件,来对状态空间进行进一步约减,使该方法更加具有可拓展性.

参　考　文　献

[1] Fedoruk A, Deters R. Improving fault-tolerance by replicating agents//Proceedings of the 1st International Joint Conference on Autonomous Agents & Multiagent Systems. Bologna, Italy, 2002: 737-744

[2] Yang J, Chen T, Wu M, et al. MODIST: Transparent model checking of unmodified distributed systems//Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation. Boston, USA, 2009: 213-228

[3] Guo H, Wu M, Zhou L, et al. Practical software model checking via dynamic interface reduction//Proceedings of the 23rd ACM Symposium on Operating Systems Principles. Cascais, Portugal, 2011: 265-278

[4] Hafeez A I, Faiq K, Osman H, et al. McSeVIC: A model checking based framework for security vulnerability analysis of integrated circuits. IEEE Access, 2018, 6: 32240-32257

[5] Bai G, Ye Q, Wu Y, et al. Towards model checking android applications. IEEE Transactions on Software Engineering, 2018, 44(6): 595-612

[6] Tian T, Zhang Y, Zhou Q, et al. ModelX: Using model checking to find design errors of cloud applications//Proceedings of the 2016 IEEE International Conference on Computer and Information Technology. Nadi, Fiji, 2016: 607-610

[7] Klai K, Ochi H. Model checking of composite cloud services//Proceedings of the IEEE International Conference on Web Services. San Francisco, USA, 2016: 356-363

[8] Clarke E M, Henzinger T A, Veith H, Bloem R. Handbook of Model Checking. Berlin, Germany: Springer, 2018

[9] Killian C, Anderson J W, Jhala R, et al. Life, death, and the critical transition: Finding liveness bugs in systems code//Proceedings of the 4th Symposium on Networked Systems Design and Implementation. Cambridge, USA, 2007: 18-32

[10] Ezekiel J, Lomuscio A. Combining fault injection and model checking to verify fault tolerance in multi-agent systems//Proceedings of the 8th International Joint Conference on Autonomous Agents and Multiagent Systems. Budapest, Hungary, 2009: 113-120

[11] Ezekiel J, Lomuscio A. Combining fault injection and model checking to verify fault tolerance, recoverability, and diagnosability in multi-agent systems. Information and Computation,

2017, 254: 167-194

[12] Gerth R, Peled D, Vardi M Y, et al. Simple on-the-fly automatic verification of linear temporal logic//Proceedings of International Symposium on Protocol Specification, Testing & Verification Warsaw, Poland, 1995: 3-18

[13] Jensen K, Kristensen L M, Wells L. Coloured Petri nets and CPN tools for modelling and validation of concurrent systems. International Journal on Software Tools for Technology Transfer, 2007, 9(3-4): 213-254

[14] Jensen K, Kristensen L M. Coloured Petri Nets. Modelling and Validation of Concurrent Systems. Berlin, Germany, 2009

[15] Leesatapornwongsa T, Gunawi H S. SAMC: A fast model checker for finding heisenbugs in distributed systems (demo)// Proceedings of the 2015 International Symposium on Software Testing and Analysis. Baltimore, USA, 2015: 423-427

[16] Leesatapornwongsa T, Hao M, Lukman J F, et al. SAMC: Semantic-aware model checking for fast discovery of deep bugs in cloud systems//Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation. Broomfield, USA, 2014: 399-414

[17] Flanagan C, Godefroid P. Dynamic partial-order reduction for model checking software//Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Long Beach, USA, 2005: 110-121

[18] Godefroid P. Partial-order methods for the verification of concurrent systems — An approach to the state-explosion problem. Lecture Notes in Computer Science, 1996, 1032

[19] Anand V. Dara: Hybrid model checking of distributed systems//Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Lake Buena Vista, USA, 2018: 977-979

[20] Clarke E M, Fehnker A, Jha S K, et al. Temporal logic model checking. Handbook of Networked and Embedded Control Systems, 2005: 539-558

[21] Bellettini C, Camilli M, Capra L, et al. Distributed CTL model checking using MapReduce: Theory and practice. Concurrency and Computation: Practice and Experience, 2016, 28(11): 3025-3041

[22] Clarke E M, Emerson E A. Design and synthesis of synchronization skeletons using branching time temporal logic. Logic of Programs: Workshop, New York, USA, 1981: 52-71

[23] Clarke E M, Emerson E A, Sistla A P. Automatic verification of finite state concurrent system using temporal logic specifications: A practical approach//Proceedings of the Conference Record of the Tenth Annual Symposium on Principles of Programming Languages. Austin, USA, 1983: 117-126

[24] Emerson E A, Halpern J Y. "Sometimes" and "not never" revisited: On branching versus linear time temporal logic. Journal of the ACM, 1986, 33(1): 151-178

[25] Katoen J P. Concepts, algorithms, and tools for model checking, 2021. https://www.researchgate.net/publication/228590811_Concepts_algorithms_and_tools_for_model_checking

[26] Gammie P, Meyden R V D. MCK: Model checking the logic of knowledge//Proceedings of the 16th International Conference on Computer Aided Verification. Boston, USA, 2004: 479-483

[27] Lomuscio A, Qu H, Raimondi F. MCMAS: A model checker for the verification of multi-agent systems//Proceedings of the 21st International Conference on Computer Aided Verification. Grenoble, France, 2009: 682-688

[28] David A, Larsen K G, Legay A, et al. UppaalSMC tutorial. International Journal on Software Tools for Technology Transfer, 2015, 17(4): 397-415

[29] Nigro C, Nigro L, Sciammarella P F. Model checking knowledge and commitments in multi-agent systems using actors and UPPAAL//Proceedings of the European Conference on Modelling and Simulation. Wilhelmshaven, Germany, 2018: 136-142

[30] Nigro L, Sciammarella P F. Statistical model checking of distributed real-time actor systems//Proceedings of the 21st IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications. Rome, Italy, 2017: 188-195

[31] Nigro L, Sciammarella P F. Statistical model checking of multi-agent systems//Proceedings of the European Conference on Modelling and Simulation. Budapest, Hungary, 2017: 11-17

[32] Guerraoui R, Yabandeh M. Model checking a networked system without the network//Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation. Boston, USA, 2011: 225-238

[33] Simsa J, Bryant R, Gibson G. dBug: Systematic evaluation of distributed systems//Proceedings of the 5th International Workshop on Systems Software Verification. Vancouver, Canada, 2010: 3-12

[34] Yabandeh M, Knezevic N, Kostic D, et al. CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems//Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation. Boston, USA, 2009: 229-244

[35] Gunawi H S, Do T, Hellerstein J M, et al. Failure as a Service (FaaS): A cloud service for large-scale, online failure drills. University of California, Berkeley, USA, 2011

[36] Gunawi H S, Do T, Joshi P, et al. FATE and DESTINI: A framework for cloud recovery testing//Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation. Boston, USA, 2011: 238-252

[37] Joshi P, Gunawi H S, Sen K. PREFAIL: A programmable tool for multiple-failure injection//Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications. Portland, USA, 2011: 171-188

[38] Basiri A, Behnam N, Rooij R D, et al. Chaos engineering. IEEE Software, 2016, 33(3): 35-41

[39] Joshi P, Ganai M K, Balakrishnan G, et al. SETSUDO: Perturbation-based testing framework for scalable distributed systems//Proceedings of the ACM SIGOPS Conference on Timely Results in Operating Systems. Farmington, USA, 2013: 1-14

LU Chao-Yi, M. S. His reseach interests include model checking and software testing.

NIE Chang-Hai, Ph. D., professor, Ph. D. supervisor. His reseach interests include software testing and software quality assurance.

CHEUNG S. C., Ph. D., professor, Ph. D. supervisor. His reseach interest is software engineering.

Background

Model checking is one of the methods in formal verification, which is used to ensure the correctness of the behavior of software and hardware systems. It is difficult for traditional software testing to verify a software which aims to find Bugs as soon as possible, rather than to prove the absence of Bugs. Because model checking needs to consider all possible behaviors of the system, it will bring the problem of state explosion. As the result of it, the application of model checking in the real world is often restricted. Existing model checkers rely on random and exhaustive strategies to inject faults into the system to be verified, bringing many disadvantages. Random fault-injection finds Bugs randomly; exhaustive fault-injection will cause the problem of the state space explosion, which limits the total number of faults that can be injected.

In this work, we propose a new way to address how to combine model checking with fault-injection technology, and realize the corresponding tool LTMC using DFS-based method with space reduction strategies to explore and verify the entire space of the system to be verified to comprehensively guarantee the correctness and fault tolerance of the software system. Also, we propose the concept of critical transition which purposefully specify the fault-injection point, rather than relying on random or exhaustive strategies. In order to solve the problem of space explosion, this work proposes a peer reduction strategy PRP based on the role of system nodes. We achieve the solution to solve the verification of the correctness and fault tolerance of distributed system protocols with a new state space reduction strategy, which cannot be done by the traditional model checking methods and software testing methods.

This research is supported by the National Key R&D Program of China (2018YFB1003800).