分布式容错计算描述语言及其应用研究

蔡媛媛 赵致琢
(厦门大学信息科学与技术学院 厦门 361005)

摘 要 在设计分布式容错系统的架构时,不仅要控制所有组件正常运行时的标准系统活动,还要控制某个组件失效时的情形。在传统的分布式容错系统设计中,这两者的模块实现往往具有紧密的耦合性,这为大型分布式容错系统的理解、设计、开发与维护增加了难度。为了解决这个问题,提出了一种新的方法,参考 Hoare 的通信顺序进程理论,使用维也纳定义语言元语言来定义这样一种描述语言:它不仅能够描述分布式计算的并发现象,还能够独立刻画系统的容错行为。这种解决方案体现了现代编程语言走向抽象化的必然趋势,也为分布式容错计算研究领域的发展提供了一种新的思路。

关键词 分布式容错计算,FTDL,VDL
中图法分类号 TP302.8 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2016.5.027

Research on Fault Tolerant Description Language and its Application for Distributed Computing

CAI Yuan-yuan ZHAO Zhi-zhuo
(School of Information Science and Engineering, Xiamen University, Xiamen 361005, China)

Abstract It is essential to control system actions in the condition that every component functions normally, as well as in the environment that some component gets failed, when designing an architecture of a distributed fault-tolerant system. Designs for the both situations are generally tightly coupled, which increase the difficulty of understanding, designing, developing, and maintaining a distributed fault-tolerant system which may be large-scale and may provide strong fault tolerance. To settle this problem, a new method was proposed in this paper, which defines a description language by the Vienna definition language on the basis of Hoare's theory of communicating sequential processes. It can describe not only the property of concurrency in distributed computing but also how a system tolerates failure. More importantly, it seems to indicate that programming languages are doomed to be more abstract in higher level in future, and it may provide a new idea in the field of distributed fault-tolerant computing.

Keywords Distributed fault-tolerant computing, FTDL, VDL

1 引言

随着银行系统、核电站控制系统、医院自助挂号系统、飞行控制系统、火车票网上订票系统、卫星控制系统等系统的开发和应用,分布式系统的容错能力逐渐成为系统设计必须考虑的一个重要因素。自20世纪80年代以来,不论是从故障、差错以及失效三者之间的关系出发,还是从失效的种类、算法的时序模型、体系结构等因素出发,分布式容错计算领域的研究都取得了很大的进展。然而,系统容错功能的设计与系统正常模块的设计之间往往具有很高的耦合性,这对于系统容错功能的设计、扩展以及系统的开发者或维护者之间的交流是不利的:一方面,系统的局部修改和维护非常困难,很容易引起新的故障或失效;另一方面,系统开发者需要耗费较多精力分析系统(原有)的架构。

为了解决这个问题,提出了一种用于刻画计算系统行为的容错描述语言(Fault Tolerant Description Language, FTDL),使系统的开发和维护人员可以用其进行系统规范描述方面的交流,而将实际的故障、失效检测、容错响应等工作交由描述语言的解释器来完成。从长远的角度来看,这种语言可作为系统开发语言的扩展部分,其设计模式有助于缩短系统开发周期,提高系统的研究开发效率。

2 相关工作

分布式容错计算领域需要面对诸多问题,如时间和全局状态问题、异步/同步协调机制、分布式事件和断言性质监控、分布式容错应激机制设计等。每个问题都可能涉及容错功能的设计,而容错功能的实现往往依赖于其它问题的实现,这实质上是因为实现容错的关键技术包括冗余技术和避错技术。例如,失效恢复技术要求周期性地保存分布式系统的形态(检查点的系统状态),通过将系统还原到最近保存的形态的方式从处理器失效中恢复。再如,失效的检测总是涉及系统形态上的谓词的检测。分布式计算领域的基本问题已经在很多文

到稿日期:2016-01-19 返修日期:2016-03-21 本文受福建省自然科学基金项目(A030007)资助。
蔡媛媛(1989—),女,硕士生,主要研究方向为计算模型与分布式算法,E-mail:cyycst@stu.xmu.edu.cn;赵致琢(1957—),男,教授,主要研究方向为计算模型与分布式算法、逻辑程序设计基础、计算机教育,E-mail:zzzhao@xmu.edu.cn。

献和著作中被提出和解决。

形式语言和编程语言都可用于描述一个分布式计算，语言的形式语法和语义是语言的必要组成部分。形式语义包括操作语义、指称语义、公理语义、代数语义。本文关注两种语言：通信顺序进程理论使用的形式语言和维也纳定义语言。

通信顺序进程（Communicating Sequential Processes，CSP）是为解决具有非确定性质的并发现象而提出的进程代数理论，是描述并发系统中通过消息交换进行交互的通信实体行为的一种形式语言。该理论能够准确刻画分布式系统中的并发性、不确定性以及通信行为。

维也纳定义语言（Vienna Definition Language，VDL）是定义程序语言的一种元语言。它使用操作语义，并且使用树结构作为数据结构。在该语言的发展过程中，出现了一种用于描述和开发顺序程序软件系统的形式化方法——维也纳开发方法（VDM）。它具有多种扩展形式，如 VDM 规范语言（VDM Specification Language，VDM-SL）、VDM ＋ ＋ 和 VDM-RT。VDM＋＋一般用于刻画面向对象的并发系统；VDM-RT 则用于描述和分析实时嵌入式分布式系统。关于上述扩展语言的主要语法形式，有学者在近几年内给出了结构化操作语义（Structure Operational Semantics，SOS）形式的语义定义。然而，目前尚未有研究工作针对分布式容错计算设计某种程序语言，或者扩展已有的某种程序设计语言，或者针对 VDM 中的异常处理语句给出形式化的语义定义。

综上所述，设计和定义一种用于描述分布式容错计算的程序语言至关重要。为了减少对语言中很多语言成分的过多关注，本文借鉴 VDM 语言已有的语法形式，使用 VDL 元语言重新定义某些语法部分（如 duration 语句）的语义，同时，修改、增加一些语言成分的定义，如借鉴远程过程调用（Remote Procedure Call，RPC）的语义，修改远程同步或异步调用的语法形式和语义定义；在系统中设置多种默认失效类型，增加对默认失效类型的监控和处理的描述。

3 描述语言的设计

本节主要介绍系统的并发性语义、duration 语句的语义定义、操作调用语句的语法和语义定义，以及虚拟差错变量的监控部分的设计。由于形式抽象语法定义和语义定义会占据相当大的篇幅，本文尽可能给出非形式的解释。另外，假设读者了解通信顺序进程理论，已经掌握维也纳定义语言以及维也纳开发方法的语言。

3.1 并发

在 VDL 语言中，解释器总是解释位于系统状态的控制分量（control component）下的指令。根据 CSP 理论，任何一个客体都可被抽象为一个进程，客体所处的环境也不例外。结合两者，可以将系统状态下的控制分量当作环境这个进程的指令。整个系统的进程分布在 3 个层次上：系统级的 CPU 对象或 channel 对象、CPU 级的 process 对象或 channel 对象，以及 process 级的 thread 对象。与 VDM-RT 语言不同的是，实际执行的对象单元除了可以是 thread 对象，还可以是 channel 对象。每个层次的对象之间的关系是并发。

根据 CSP 理论中的并发语义，对于不需要协同的情形，只需要依次从 process 级、CPU 级、系统级进行对象的不确定性选择即可从低级到高级确定环境调度的下一个执行对象。因此，每个层次的状态都具有 s-nextth 分量或者 s-nextp 分量用于产生每个层次下一个被调度的对象，并且使用 s-cur-t 分量或者 s-cur-p 分量记录该对象。为了处理 thread 对象之间需要协同的情形，每个 thread 对象的状态都具有 t-status 分量，其值可能是 RUNNING、RUNNABLE、WAITING 或 TERMINATED。只有 t-status 值为 RUNNABLE 的对象才能够被调度；该值为 RUNNING 的对象正在被解释；该值为 WAITING 或者 TERMINATED 的对象不能被调度。本文定义的 channel 对象只处理通信时间的标记，属于不需要协同的情形。

另外，除了上述并发语义隐含的不确定性外，VDL 语言自身包含了另一种不确定性：解释器可以选择当前控制分量（也称控制树）上的多个终端顶点中的任何一个顶点上的指令作为下一条被解释的指令。

3.2 duration 语句

在描述语言 FTDL 中，唯一包含时间语义的成分是 duration 语句，它规定了语句的执行时间。此处定义的语义与 VDM-RT 中的是一致的，即语句对系统形态的影响必须在规定时间过去之后生效。

由于 duration 语句的定义包含一个可能嵌套 duration 语句的语句列表，并且 duration 语句只可能出现在 thread 对象的代码中，因此在每个 thread 对象的状态中增加了一个结构为堆栈的 t-duration 分量。解释器在解释每个 duration 语句前计算该语句的剩余时间和设定时间，并将这两个记录入栈，同时保存当前的状态到 thread 对象的 t-pending 分量中；在完成对每个 duration 语句的语句列表解释时，将 t-pending 分量的值删除，对 t-duration 分量执行出栈操作和更新当前值操作，最后对当前对象的 CPU 时间进行更新。

对于 t-pending 分量的使用，还需要作进一步的说明。如果没有其它对象访问 duration 语句修改的状态值，则无需任何操作，只需顺序执行即可；否则，其它对象对某个待更新状态的访问蕴含着调度过程的执行。如果调度过程在对某个 duration 语句的解释完成之前（即调度之前的对象的 t-pending 分量非空），就需要将 t-pending 分量保存的状态（duration 语句执行之前的状态）中的相关分量与当前对象的状态分量（最新状态）的相应分量进行交换，从而保证在规定时间被使用完之前相关的状态分量仍然是 duration 语句未执行之前它们所具有的值。这部分过程由用于解释调度过程的宏指令 doSwitch 的展开式中的指令 try-doSwitch 负责解释，主要由其展开式中的指令 checkThreadStatus 完成，其 VDL 定义如下所示。

```
doSwitch =
  try-doSwitch(b,sid,pchid,tid,sp);
    b:check-p-ve-guard;
      update-cur-p(1);//更新 sys-cur-p 分量
        store-sys-c;//保存 sys-c                    (I1)
          sid:pass-v(sys-cur-p(ξ)),
          pchid:pass-v(cpupch_cur),
          tid:pass-v(t_cur),
          sp:pass-v(ξ_p)
try-doSwitch(t,sid,pchid,tid,sp)=
  convert(t,bool)→
    update-cur-sc(sid,pchid,tid);//更新各层次的 s-c 分量   (I2)
```

checkThreadStatus(sid,pchid,tid,sp);

T→doSwitch

checkThreadStatus(sid,pchid,tid,sp)=

(sid=cpu_{cur} ∧ pchid=$cpupch_{cur}$ ∧ tid=t_{cur})→null

(¬is-Ω(ξ_{cpuid}) ∧ ¬is-Ω(ξ_{pid}) ∧ ¬is-Ω(ξ_{tid})

∧ ¬is-Ω(ξ_{cpu}) ∧ ¬is-Ω(ξ_{p}) ∧ ¬is-Ω(ξ_{t})) →

ChangeThreadStatus(cpu_{cur},$cpupch_{cur}$,t_{cur},RUNNING);

check-post-thread; //tcur 线程要被换进来时的处理

check-pre-thread(sid,pchid,tid,sp)//tid 线程要被换出时的处理

(¬is-Ω(ξ_{cupid}) ∧ ¬is-Ω(ξ_{pid}) ∧ ¬is-Ω(ξ_{tid})) → (I3)

check-pre-thread(sid,pchid,tid,sp)

(¬is-Ω(ξ_{cpu}) ∧ ¬is-Ω(ξ_{p}) ∧ ¬is-Ω(ξ_{t})) →

ChangeThreadStatus(cpu_{cur},$cpupch_{cur}$,t_{cur},RUNNING);

check-post-thread

(¬is-Ω(ξ_{cpu}) ∧ ¬is-Ω(ξ_{p}))→//当前调度的是 process 中的 ctl-t

ChangeThreadStatus(cpu_{cur},$cpupch_{cur}$,t_{cur},RUNNING);

(¬is-Ω(ξ_{cpu}) ∧ is-n(cpupch$_{cur}$) ∧ ¬is-Ω(ξ_{cpuid}) ∧ is-n(pchid))→

ChangeSysThreadStatus(cpu_{cur},$cpupch_{cur}$,RUNNING),

ChangeSysThreadStatus(sid,pchid,RUNNABLE)

(¬is-Ω(ξ_{cpu}) ∧ is-n(cpupch$_{cur}$))→ //新 id 是 idle-n、rpc-n、ctl-n

ChangeSysThreadStatus(cpu_{cur},$cpupch_{cur}$,RUNNING)

(¬is-Ω(ξ_{cpuid}) ∧ is-n(pchid))→ //旧 id 是 idle-n、rpc-n、ctl-n

ChangeSysThreadStatus(sid,pchid,RUNNABLE)

T→null//这步不会执行,方便后续扩充

其中,ξ_{cpuid}=sid · sys-cpu(ξ),ξ_{pid}=pchid · s−ps(ξ_{cpuid}),ξ_{tid}=tid · s−t(ξ_{pid})。

3.3 操作调用语句

操作调用分为 4 种:本地同步调用、本地异步调用、远程同步调用和远程异步调用。其抽象语法定义与 VDM 的类似,如下所示。

(A1) is-proc-call = (⟨s-sync:is-async-id⟩, ⟨s-target:is-target⟩, ⟨s-name:is-cn⟩, ⟨s-arg-list:is-expr-list⟩)

(A2)is-target=(⟨s-v:is-id⟩, ⟨s-pid:is-id⟩, ⟨s-tid:is-n⟩)

(A3)is-cn=(⟨s-pid:is-id⟩, ⟨s-opid:is-id⟩)

(A4) is-proc-attr = (⟨s-async:is-async⟩, ⟨s-tw-check:is-bool⟩, ⟨s-param-list:is-id-list⟩, ⟨s-type-list:is-id-list⟩, ⟨s-ret-type:is-id⟩, ⟨s-body:is-st-list⟩, ⟨s-pre:is-expr⟩, ⟨s-post:is-expr⟩, ⟨s-mutex:is-expr⟩)

(A5)is-sync-decl=(⟨s-pn:is-id⟩, ⟨per-g:is-expr⟩)

(A6)is-ve-guard=(⟨s-ve-b:is-bool⟩, ⟨s-ve:is-id⟩, ⟨s-op-list:is-id-list⟩, ⟨s-handler-proc:is-proc-call⟩)

注意到,操作调用总是与一个操作标识符相关。程序中声明的每个操作(operation)的定义都会被保存在一个满足谓词 is-den 的标记表中。除了操作标识符外的操作语法定义满足谓词 is-proc-attr。注意,操作的语法声明并不包含 s-tw-check 分量(component)和 s-mutex 分量;s-tw-check 分量的值由程序中的语句设定,s-mutex 分量的值取决于满足谓词 is-sync-decl 的同步限制声明。与操作相关的还有满足谓词 is-ve-guard 的虚拟差错变量的监控声明,这些将在下一节中介绍。

操作调用语句将由指令 int-n-proc 解释。如果该调用涉及的操作的同步限制表达式的值为假,则修改当前 thread 对象的状态,然后执行指令 doSwitch,并将其控制分量重新置为指令 int-n-proc,它将在下次被调度时重新展开;否则,该操作的同步限制表达式的值为真。此时,如果该操作的前置条件为假,解释器认为这是一种失效,并将根据 s-tw-check 分量的值决定是否对这种失效进行二次检测;如果该操作的前置条件为真,或者不存在前置条件,则解释器会检查与该操作相关的虚拟差错变量的值以判断该操作是否具备发生的条件,然后依次执行以下 3 条指令:save-state 指令、int-extend-proc-call 指令和 check-post-g 指令。第二条指令的展开式中存在一个比较重要的指令——int-proc-call 指令。该指令根据操作调用的抽象语法定义中的成分判断调用的种类,然后将当前正在被解释的指令替换为以下 4 种指令之一:int-proc-remote-sync 指令、int-proc-local-sync 指令、int-proc-remote-async 指令和 int-proc-remote-async 指令。

不论是本地调用还是远程调用,异步调用的语义总是被解释为由当前 thread 对象在所处 process 环境中创建一个新的 thread 对象,由新对象处理这个特殊的同步调用。因此,同步调用的语义是关键所在。本地同步调用的语义与 VDM 的 SOS 语义是一致的。本文定义的远程同步调用的语义借鉴了远程过程调用(Remote Procedure Call,RPC)的语义。如果当前系统默认设置了缓存机制,并且系统保存有该远程调用的返回值,则直接返回该值;否则,标记该调用的状态为 UNCOMPLETED,执行负责保存调用信息到相关状态分量的指令 int-rpc-proxy,然后修改当前 thread 对象的 t-status 分量,触发专用于处理 RPC 的 process 对象,最后执行 doSwitch 指令,并重置当前指令为等待某个值返回,或者等待调用完成的指令 Wait-Return。

专用于处理 RPC 的 process 对象负责将操作调用封装为调用消息,并将其赋值给 channel 对象的相关分量。它还负责处理被各 channel 对象标记为已到达的消息。如果某个消息被标记需要返回确认,则设置相关 channel 对象的相关分量为返回的确认消息。如果某个消息是调用类型的,则在目的 process 对象中创建一个新的 thread 对象,用于执行与该消息相关的本地同步调用,并且为该调用设置一个计时器线程。如果某个消息是返回结果类型的(这种消息由执行 RPC 的 thread 对象生成),则将结果值返回给 RPC 的调用者。最后,如果某个消息是确认类型的,则设置调用的确认标志为真。消息的确认、副本过滤和超时重传都由该 process 对象隐式处理,其中,超时重传实际是由计时器线程完成的。

3.4 虚拟差错变量的监控部分

解释器在每个 CPU 对象中都设置了一个控制 process 对象,在每个程序定义的 process 对象中都设置了一个控制 thread 对象。它们都用于处理虚拟差错变量的监控。解释器默认设置了 14 种虚拟差错变量,分别负责标记以下失效:

(1)变量的赋值违背了其不变量定义;

(2)程序声明的类型无定义;

(3)只读变量被多次赋值;

(4)声明有返回值的调用实际没有返回任意值;

(5)调用的输入参数数目不正确;

(6)由于后置条件不满足导致的函数调用失效;

(7)发生在 duration 语句上的失效;

(8)未知类型失效:操作调用或者函数调用的前置条件不满足或者程序中调用 error 语句;

(9)由于后置条件不满足导致的本地操作调用失效;

(10)某个失效没有被程序声明处理;

(11)不存在可用 channel 对象用于通信;

(12)不存在 RPC 指定的 process 对象或操作;

(13)消息无确认:可能是崩溃失效;

(14)远程调用无返回值回复:可能是网络失效。

其中,前 10 种由控制 thread 对象处理,后 4 种由控制 process 对象处理。如果前 9 种失效中的某一种失效发生,但是程序并没有声明其处理操作,则它将成为第(10)种失效,会被当前 process 对象中的其它 thread 对象处理。失效发生时正在执行的调用或者所处的 thread 对象标识符、process 对象标识符、CPU 对象标识符会在标记失效时保存在与失效信息相关的状态分量中。用于处理失效的操作必须将该失效对应的虚拟差错变量设置为假。这种使用虚拟差错变量的机制普遍出现在分布式容错计算领域的理论研究中。

借鉴这种监控机制,解释器为每个 process 对象设置了一个默认使能变量"__ve-process-switch",其值默认为真,当程序没有为作用于 CPU 级别的失效声明处理 process 对象或者处理操作时,解释器默认将当前 process 对象的使能变量赋值为假,即关闭当前 process 对象。该方案主要借鉴了故障弱化(graceful degradation)的思想。

另外,定义的解释器给出了异常处理语句的语义定义,它们的作用与上述前 9 种失效处理机制是类似的。为了解释其语义,考虑到语句可能是嵌套的,解释器为每个 thread 对象的状态增加了一个结构为堆栈的 t-call-d 分量,用于记录每一层发生的异常类型和异常的调用。若最底层的异常在某一层得到处理,则异常堆栈将被置空;否则,就会被当前正在执行的 thread 对象标记为上述第(10)种失效。

4 描述语言的应用

本节将通过哲学家问题这一实例说明如何用 FTDL 容错描述语言建立一个分布式计算的规范,该实例如下所示。

```
EXAMPLE I
System fault_Tolerant_System
    CPU cpu1 = new CPU (⟨22E6⟩); CPU cpu2 = new CPU
    (⟨22E6⟩);CPU cpu3 = new CPU(⟨22E6⟩); CPU cpu4 = new
    CPU(⟨22E6⟩);CPU cpu5 =new CPU(⟨22E6⟩);
    BUS bus1 = new BUS (⟨72e3⟩, (cpu1, cpu2, cpu3, cpu4,
    cpu5));
    Process p1 =new Phe();Process p2=new Phe();Process p3 =
    new Phe();Process p4=new Phe();Process p5 =new Forks();
    Process p6=new Init();
    cpu1. deploy(p1); cpu2. deploy(p2); cpu3. deploy(p3);
    cpu4. deploy(p4); cpu5. deploy(p5); cpu5. deploy(p6);
end fault_Tolerant_System
Class Phe
  instance variables
    id:nat;
  operations
    public setId:nat⇒()
    setId(num) == id :=num;
  op-guards
    __ve-MSG-NoAck-Failure(true, ⟨tryIgnore⟩,tryIgnore(7))
    __ve-RPC-NoReply-Failure(true,⟨tryIgnore⟩,tryIgnore(8))
    __ve-UnknownFailure(true, ⟨SwitchOff⟩, SwitchOff());
  sync
    per Think⇒#fin(setId)>0 //wait until a "setId" finishes
  threads
    thrPhe:durations(
      dcl rid:nat :=(id+1) mod 5;
      if rid=0 then(rid :4;id :1;) //change direction
        while true do(
          duration(30)(Think();)
          while not p5. tryGetFork(id) do
            duration(20)(skip)
          p5. GetFork(id);
          while not p5. tryGetFork(rid) do
            duration(20)(skip)
          p5. GetFork(rid);
          duration(10)( DinnerTime();)
          p5. PutFork(rid);p5. PutFork(id);)
    )
end Phe
Class Forks
  instance variables
    pforks:map nat to bool :=
    {1 ↦false,2 ↦false,3 ↦false,4 ↦false};
  operations
    public tryGetFork:nat⇒bool
    tryGetFork(ef) ==
      if pforks(ef)=true then return false;
      else return true;
    public GetFork:nat⇒()
    GetFork(ef)==
      pforks(ef):=true;
      pre pforks(ef)=false;
    public PutFork:nat⇒()
    PutFork(ef)==pforks(ef):=false;
  op-guards //may try again with tryGetFork
    __ve-UnknownFailure(true, ⟨TryAgain⟩, TryAgain());
  sync
    per GetFork⇒#active(GetFork)<5
    per PutFork⇒#fin(GetFork)>#fin(PutFork)
end Forks
```

实例的代码涉及前面提及的 3 个层次的对象。在该实例中,4 个哲学家都是 process 对象,p_5 是叉子 process 对象。它们分别被分派到 5 个不同的 CPU 对象中。CPU 对象之间通过 bus_1 对象连通。该例子只涉及哲学家对象与 p_5 之间的通信。对叉子的互斥操作由 p_5 提供。然而,操作"GetFork"的互斥条件只能保证第 5 个被激活的调用必须等待前 4 个活动调用中某一个调用完成,不能保证前 4 个调用的输入参数各不相同。这里,可以选择限制该操作的前置条件。如果在执行相关调用前,该前置条件不满足,则解释器认为"UnknownFailure"类型的一个失效正在发生,其对应虚拟差错变量"__ve-UnknownFailure"将被隐式地赋值为真。解释器默认设置的控制 thread 对象将触发调用"TryAgain()"的执行来处理该失效。在哲学家的虚拟差错变量监控声明中,还可能存在与远程调用相关的其它两种失效,这两种失效由解释器默认设置的控制 process 对象监测。

(下转第 168 页)

niques of Anticipatory Monitors for Parameterized LTL[J]. Journal of Software,2010,21(2):318-333

[25] SNU Real-Time Benchmarks[OL]. http://www.cprover.org/goto-cc/examples/snu

[26] Zee K, Kuncak V, Taylor M, et al. Runtime checking for program verification[C]//Proceedings of the 7th International Conference on Runtime Verification(RV'07). Berlin, Heidelberg: Springer-Verlag,2007:202-213

[27] Zhou W, Sokolsky O, Loo B T, et al. DMaC: Distributed Monitoring and Checking[M]//Runtime Verification(RV). 2009:184-201

[28] Navabpour S, Bonakdarpour B, Fischmeister S. Path-aware time-triggered Runtime Verification [M] // Runtime Verification (RV2012). 2012:184-201

[29] Drusinsky D. On-line monitoring of metric temporal logic with time-series constraints using alternating finite automata[J]. JUCS,2006,12(5):482-498

[30] Basin D, Klaedtke F, Miiller S, et al. Runtime monitoring of metric first-order temporal properties[J]. Best Practive & Research Clinical Obstetrics & Gynaecology,2010,22(5):899-916

[31] Bodden E, Lamp. Clara: Partially Evaluating Runtime Monitors at CompileTime[M] // Runtime Verification. Springer Berlin Heidelberg,2010:74-88

[32] Kim C H P, Bodden E, Batory D, et al. Reducing configurations to monitor in a software productline[M] // Runtime Verification. Springer Berlin Heidelberg,2010:285-299

(上接第 149 页)

实例中,用于处理失效的操作都没有给出,它们不是本文讨论的重点。针对不同用途的分布式系统,系统设计开发者可以在此基础上设计默认的相关处理机制,帮助实现系统容错功能的半自动化甚至全自动化。注意,实例中的虚拟差错变量监控声明都是用于监控某个虚拟差错变量为真的情况,而实际上,还可以监控虚拟差错变量为假的情况。在后一种情况下,"某个虚拟差错变量为假"变成了相关指定操作的一个前置条件。这种机制能够防止该操作在某种失效发生后被执行,除非该失效被修复。

结束语 本文针对传统分布式容错系统设计在开发大规模容错系统或者容错能力较强的分布式系统时存在的不足,设计了一种抽象的容错描述语言 FTDL。该语言在解释器层面处理故障、差错或者失效的定义、检测和修复。这种解决方案试图在更抽象的层次以比面向过程程序设计语言和面向对象程序设计语言更加简练的方式描述一个分布式计算的规范。从发展的眼光看,这种程序设计思想可望有效提高分布式容错系统设计的效率。

限于篇幅,本文简要阐述了新的构想、科学意义和应用前景,给出了一些语言成分及其语义描述,并用这种语言针对典型实例给出了分布式容错计算描述的规范,通报了这项研究的最新进展。有关这种语言的详细设计、语法与语义定义、多种应用,作者将另文发表。

参 考 文 献

[1] Laprie J C. Dependable computing and fault tolerance: Concepts and terminology [C]// Proceedings of the 15th International Symposium on Fault-Tolerant Computing. 1985:2-11

[2] Elena D. Fault-Tolerant Design [M]. New York: Springer, 2013:15-16

[3] Herlihy Maurice P, Jeannette M W. Specifying Graceful Degradation [J]. IEEE Transactions on Parallel and Distributed System,1991,2(1):93-104

[4] Gärtner C. FELIX, Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments [J]. ACM Computing Surveys,1999,31(1):1-26

[5] Avižienis A. Design of Fault-Tolerant Computers [C]// Proceedings of the AFIPS'67 Fall Joint Computer Conference, 1967. Washington: Thompson Books,1967:733-743

[6] Alpern B, Schneider F B. Defining liveness [J]. Information Processing Letters,1985,21(4):181-185

[7] Ajay D K, Mukesh S. Distributed Computing(Principles, Algorithms, and Systems) [M]. New York: Cambridge University Press,2008

[8] Needham R M, Herbert A J. The Cambridge Distributed System [M]. Addison Wesley International Computer Science Series, 1982

[9] Hoare C A R. Communicating Sequential Processes [M]. Prentice Hall International,2004

[10] Peter W. The Vienna Definition Language [J]. ACM Computing Surveys,1972,4(1):5-63

[11] Hoare C A R. Communicating sequential processes [J]. Communications of the ACM,1978,21(8):666-677

[12] Lamport L, Shostak R, Pease M. The Byzantine generals problems [J]. ACM Transaction on Programming Languages and Systems,1982,4(3):382-401

[13] Pierre A. The practical importance of formal semantics [M]. Liber Amicorum,1989:31-40

[14] Larsen P G, Lausdahl K, Battle N. The VDM-10 Lauguage Manual: TR-2010-06 [R]. the Overture Open Source Initiative, April 2010

[15] Lausdahl K, Coleman J W, Larsen P G. Semantics of the VDM Real-time Dialect: ECE-TR-13 [R]. Adarhus University, April 2013

[16] Birrell A D, Nelson B J. Implementing Remote Procedure Calls [J]. ACM Transaction on Computer System,1984,2(1):39-59

[17] Wang Yan-yan, Liu Wei, Wang Zhi-ming. Networked fault tolerant control for uncertain singular systems with a packet dropout[J]. Journal of Chongqing University of Posts and Telecommunications(Natural Science Edition),2013,25(3):379-383(in Chinese)
王岩岩,刘伟,汪志鸣. 数据包丢失的不确定奇异系统网络化容错控制[J]. 重庆邮电大学学报(自然科学版),2013,25(3):379-383