第7卷第10期
1996年10月

软 件 学 报
JOURNAL OF SOFTWARE

Vol. 7, No. 10
Oct. 1996

基于图文法的并发系统状态测试方法及其实现*

徐建礼　　周龙骧

（中国科学院数学研究所　北京　100080）

摘要　在并发系统的研究和开发中，迫切需要一种能正确有效地描述并发系统的动态进程互联结构、动态进程通信和进程演化行为的形式化方法以及基于这种形式化方法的并发系统动态状态的测试手段. 本文介绍一种基于图文法模型的并发系统状态测试方法，该方法与描述并发系统结构和行为的图文法模型相结合，构成了一个并发系统开发支持环境. 这一方法可根据对并发系统的状态测试要求，在并发系统的运行期自动跟踪和记录并发系统的运行状态和通信情况，使并发系统的开发者可以实时地得到并发系统的运行状态，或者在并发系统运行结束后重演并发系统的状态变化过程.

关键词　并发系统，系统测试，图文法，形式化方法，容错与同步算法.

开发并发系统软件目前面临着2个主要难题：①程序员在并发系统的程序中要具体地描述和安排系统的进程互联结构(Process Topology)及其所有可能的变化，其中包括每个进程的创建和撤消、每条通信链路的建立和撤消等. 这些问题在传统的具有静态进程互联结构的并发系统中并不难解决，但在当前的面向对象的并发系统中，进程的互联结构是动态的，进程以及进程之间的通信关系随着系统中对象的变化而动态地创建或撤消. 在现有的进程控制和进程通信机制的支持下，要想正确和高效地安排好这种动态进程结构、动态进程通信和进程到站点机上的动态映射，其难度和工作量都异常之大；②对于具有动态进程互联结构的并发系统，由于其运行过程中所特有的不确定性，很难对其进行系统的跟踪和调试(Test and Debug)，如：对其中某个进程设置断点后，将会对整个系统的行为产生影响. 目前还缺乏从整个系统的高度对并发系统进行调试或状态测试的手段.

为解决上述难题，当前迫切需要一种能正确有效地描述并发系统的动态进程互联结构、动态进程通信和进程演化行为的形式化方法以及基于这种形式化方法的并发系统开发支持环境. 为此，我们设计了一个新的实用化的图文法模型，并以该模型为基础，设计实现了一个并发系统开发支持环境的原形系统(GRADECS). 在GRADECS系统中为并发系统的开发

---

* 本文研究得到国家自然科学基金、国家863高科技项目基金、中科院院长基金和中科院管理决策与信息系统实验室资助. 作者**徐建礼**，1962年生，博士，副研究员，主要研究领域为分布式数据库，并发系统. **周龙骧**，1938年生，研究员，博士导师，主要研究领域为分布式数据库系统，多媒体数据库系统.

本文通讯联系人：周龙骧，北京100080，中国科学院数学研究所

本文1995－09－22收到修改稿

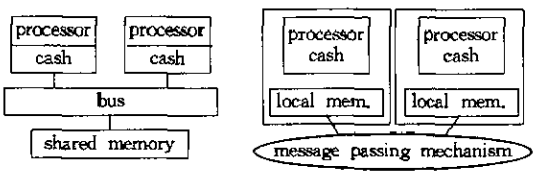
者提供了2个有力工具:基于图文法的并发系统形式化描述语言CSDL(concurrent system description language)和并发系统测试说明语言TDL(testing description language). GRADECS系统根据对并发系统的进程结构与通信结构的形式化描述(用CSDL语言说明),为其生成运行期的进程控制机制和进程通信服务机制;同时,还可根据对并发系统的状态测试要求(用TDL语言说明),在并发系统的运行期自动跟踪或记录并发系统的运行状态和通信情况,使并发系统的开发者可以实时地得到并发系统的运行状态,或者在并发系统运行结束后重演并发系统的状态变化过程.本文重点介绍GRADECS系统中的并发系统状态测试方法及其实现.

本文第1节介绍并发系统中的一些基本概念;第2节简单介绍图文法模型及其对并发系统的描述方法;第3节详细介绍基于图文法的并发系统状态测试的思想和方法;第4节介绍GRADECS系统中并发系统状态测试子系统的实现算法;第5节是总结.

1 并发系统中的基本概念

1.1 多机系统

并发系统可运行在支持多进程的单机系统结构上,但为了探讨并发系统的本质和使问题更具普遍性,我们假设并发系统运行在多机体系结构(又称为多处理机体系结构)上.



(a) 共享存储多处理机结构 (b) 分布式存储多处理机结构

图1 多机体系结构

多机体系结构又可分为共享存储和分布式存储.图1是这2种结构的示意图.我们将多机体系结构中的每台处理器(附带其局部存储器)称为一个站点(Site).本文我们主要对分布式存储结构的多机系统感兴趣.

多机体系结构上要具有支持多处理的操作系统,包括统一的多处理操作系统或支持多进程的单机操作系统与网络通信软件系统的结合.

1.2 并发系统

一个并发系统,直观地讲就是一个运行在多机体系结构上的软件系统.并发系统的基本组成部分是分散在不同站点上的进程.

并发系统可直观地定义如下[1]:一个并发系统由一组进程和一组共享对象(Shared Objects)构成,每个进程由一个顺序程序所定义,共享对象使得各进程能相互通信、相互配合以完成某些任务.

所谓共享对象可由共享存储器或通信网络实现,其功能是使并发系统内的各个进程能相互通信和相互同步.

并发系统中位于同一站点上的、对应于同一顺序程序的所有进程,构成一个集合,称为一个进程簇(Process Cluster).对于一个按模块化要求设计实现的并发系统,进程是某一功能模块的一次执行(或称为活跃),因此,我们将该功能模块所对应的进程簇称为一个功能簇(Function Cluster).一个并发系统在其运行期可看作是由若干功能簇构成的集合.

1.3 进程通信

为了在处理相关的计算任务时相互配合,进程之间必须相互通信和同步.所谓进程通

信,就是使一个进程去影响另一个进程的执行.通信可以通过使用共享变量或消息传递(Message Passing)来实现.

在进程通信的过程中,信息的发送与收到(或写与读)之间有一个不可忽略的时间延迟.这一延迟使得某项信息的接收进程只能了解该信息的发送进程的一个过时的状态.这一因素使得在并发系统中无法准确地了解各进程的当前状态,造成并发系统中进程管理和跟踪的困难.

2　使用图文法描述并发系统

2.1　为什么要使用图文法

从80年代以来,由于并发系统的研究与开发的需要,并发系统的形式化描述成了人们研究的一大热门课题,并且产生了许多并发系统的描述工具.图文法是一个重写(Rewriting)系统,它描述图的构成及其变化、图中各个子部分之间的联系及其变化,它通过引用一个有限图文法规则集中的规则(图文法产生式),从一个初始图推出一系列新的图.在探寻并发系统的图形化描述方法时,我们觉得图文法是一条很好的途径.[2]

2.2　图文法模型

本节我们简单介绍图文法模型中的一些最基本概念(图文法模型的详细定义见文献[2,3]),然后用一个例子来说明怎样用图文法描述一个并发系统.

2.2.1　图

在描述并发系统时,图文法模型中所采用的图的形式要能够反映出并发系统的结构特点.在我们的图文法模型中,图中包含3类元素:节点(Node)、端口(Port)和节点与端口之间的有向边(Arc),它们分别代表了并发系统中的组成元素(程序模块或进程)、元素上的消息通信端口以及元素之间的依赖关系和通信联系.

图2是一个图的例子.在图中,节点由一个中间标有节点名的大圆表示,端口由附在节点周围的小圆表示,每个小圆中标有端口名,还有一些从节点指向端口的有向边.

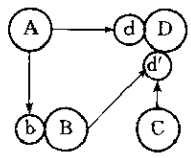


图2　1个图的例子

2.2.2　图文法产生式与图的重写

图的重写是通过引用图产生式将一幅图变换成另一幅图.为了避免子图的识别与替换等一类复杂问题,我们采用节点标识控制的(Node-Label Controlled)图文法[4],简记为NLC图文法.在NLC图文法中,每次重写是用一个子图来置换一个节点,被置换节点的节点标识决定了使用哪一条产生式.

一个产生式具有如下形式:$p:L\rightarrow B,Em$,其中 p 为该产生式的唯一标识;L 为一节点名,是产生式的左部或称为目标(Goal);B 是用来替换 L 的一个图,称为 body graph;Em 是嵌入说明,它说明 B 替换 L 后如何嵌入到 L 原来所在的图中,Em 由标有嵌入表达式(Edge-End Embeding Expression)的槽(Sockets)以及从这些 *sockets* 到 B 中端口或从 B 中节点到这些 *sockets* 的有向边组成;B 和 Em 一起称为产生式的右部.

对于图文法产生式也采用直观的图形表示法,也就是将产生式中的 B 和 Em 表示成一个特殊的图.在上述定义中,Em 中的每一个 *socket*(在图形表示中为一个方框)中所标记的

嵌入条件表达式(简记为 E^4)表示的是 B 中的节点或端口可与原图中的哪些端口或节点相连. 在并发系统中,嵌入说明用以描述新产生的进程与系统中其他进程的通信和同步关系. 文献[2,3]给出了 E^4 的具体定义.

图文法中最基本的动作是直接的一步推演,称为重写. 重写就是引用某一条图文法产生式来对被重写的图进行操作,以产生一幅新图.

在图 3 给出一个图文法产生式及引用产生式进行重写的例子. 在这个例子中,产生式 p 的作用是将节点 A 替换成由节点 A 和节点 B 组成的 *bodygraph*,p 一共被引用了 2 次. 在 p 的嵌入说明部分中一共有 2 个标有 E^4 表达式的 *sockets*,第 1 个 *socket* 中的 E^4 表达式为"a",其结果是在原图中与节点 A 的端口 a 有边相连的节点的集合,在这里为{D};第 2 个 *socket* 中的 E^4 表达式为"*filter*(*OUT*, x. *node_name*=C)",其结果是一个端口集,它包括所有在原图中属于节点 C 且与节点 A 相连的端口,这里为{c}.

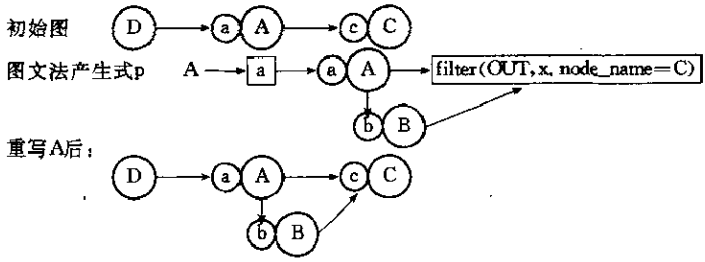


图3 图文法产生式和重写的例子

2.3 用图文法描述并发系统的实例

2.3.1 图系统与并发系统

从一幅图出发,经过一系列图文法产生式的重写,可得到一组图,我们称之为一个图系统. 1 个图系统中包括:1 幅初始图,称为 axiom graph;1 个图文法产生式集合;所有通过引用产生式集合中的产生式或产生式序列,从初始图出发所导出的图.

这里假设并发系统是按模块化的方法设计和实现的(面向对象的并发系统是最好的模块化系统). 每个并发系统都有其初始互联结构(静态结构)和动态互联结构(动态结构). 并发系统的静态结构包括一组程序模块和这些模块之间的相互依赖关系. 在面向对象的系统中,这些依赖关系表现为通信关系. 在并发系统启动后,并发系统便成为由若干功能簇(简记为 f—cluster)构成的动态集合. 并发系统的动态互联结构包括其进程结构和进程间的通信联系. 在描述一个并发系统时,我们既要描述其静态结构也要描述其动态结构.

一个并发系统的静态结构可以用一幅图来表示,这幅图称为该并发系统的 Axiom 图. Axiom 图中的节点和端口分别由程序模块名和形式端口名来标记. Axiom 图中的端口并不是真正的消息通信端口,而是用以描述各程序模块之间的各种关系. 这些相互关系在并发系统启动后,便会演化为各程序模块所对应的进程上的消息通信端口和进程之间的实际通信链路. 因此,将 Axiom 图中的端口称为形式端口.

在并发系统的运行期,每个程序模块所对应的实体是一个功能簇,因此,描述并发系统的动态结构就是描述这些功能簇. 其中主要包括:功能簇中的进程如何生成与撤消、进程结构的演化规则、实际通信链路的建立规则等. 从形式上看,一个功能簇包含下述内容:

功能簇所对应的程序模块 m;从 m 生成的所有进程的集合 P;用于表示进程间消息通

信的端口的集合 $Port$;图文法产生式集合 $Prod$,其中的产生式用来描述进程结构的变化规则和通信链路的建立规则;用于说明系统变化条件的触发事件集合 T,T 中的事件包括:某一给定类型的消息的到达、超时、站点故障、消息队列空或溢出,以及其他系统可以感知的事件;描述触发事件集合 T 与产生式集合 $Prod$ 中元素对应关系的映射 $h:T\rightarrow p(Prod)$,当 T 中的事件发生时,映射 h 规定了应该引用 $Prod$ 中的哪条图文法产生式来使并发系统发生相应的变化(注:$p(Prod)$为 $Prod$ 的幂集).

在并发系统运行期的某一时刻,并发系统的动态结构可用一个图来表示,这个图称为并发系统的结构状态图,简称为状态图.状态图可以通过对并发系统的 $Axiom$ 图引用某一条图文法产生式或产生式序列进行重写而得到.由此,我们通过一个图系统就能完整地从静态到动态地描述一个并发系统.

2.3.2　一个实例

在这一部分中我们给出一个用图文法描述并发系统的例子.为了使例子既简单明了又具有一定的使用性,我们取分布式数据库管理系统 C—POREL[5] 的系统结构中最核心的部分:事务管理子系统和数据执行子系统(关系基本机器),作为一个并发系统的实例,称之为 sample 系统.

假设 sample 系统中有 n 个站点($n\geqslant 3$),有 2 个程序模块:TM 和 RBM.TM 负责分布式数据库的分布式事务的管理;RBM 负责各站点上具体的数据操作,执行对局部数据库的各种操作.TM 的进程在每个站点上只有一个,该进程在系统初启时启动;RBM 的进程在每个站点上可以有若干个,每个 RBM 进程对应于一个事务,RBM 进程由 TM 进程来控制(启动和撤消).sample 系统的进程结构如图 4 所示.

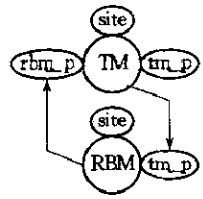


图4　sample系统的Axiom图

下面我们用图文法来描述 sample 系统.sample 系统的静态结构中只有 2 个节点:TM 和 RBM.TM 向 RBM 传递操作要求,RBM 向 TM 返回操作结果或错误代码,不同站点上的 TM 之间要相互合作来处理分布式事务的全局管理.这些关系由 sample 系统的 Axiom 图表示(图 4),Axiom 图中的形式端口说明了这 2 个模块之间的通信关系.

接下来我们描述 sample 系统的动态结构.动态结构的核心内容是用图文法产生式来描述各个 f—cluster 中进程的创建和撤消、各个进程之间通信端口和通信链路的建立规则.根据 sample 系统的具体功能,应该有以下 4 条图文法产生式:

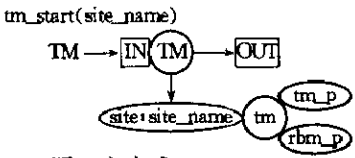


图5　产生式tm_start()

(1)tm_start(site_name)

描述在指定的站点(由 site_name 参数指明)上如何启动事务管理模块的进程,如图 5 所示.这条产生式只在系统初启时引用.新产生的进程节点 tm 从 Axiom 图中的程序模块节点 TM 继承了 2 个形式端口 tm_p 和 rbm_p,tm 进程由标记有 site_name 参数值的端口来标识.

(2)start_transaction(site_name,site_list,tid)

描述在启动一个分布式事务(由 tid 唯一标识)时,该事务的发起站点(由 site_name 参数指明)上的 tm 进程如何与参与该事务执行的所有伙伴站点(由 site_list 参数表给出)上

的 tm 进程建立通信联系，如图 6 所示.

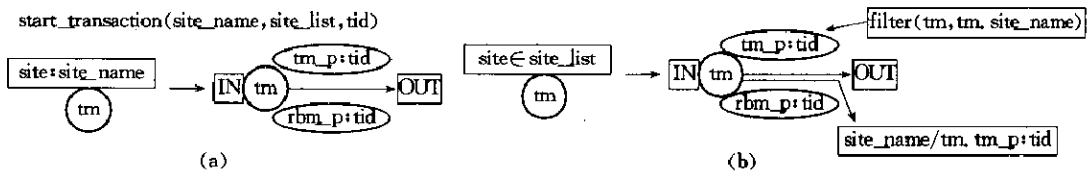


图6 产生式start_transaction()

这条产生式有(a)和(b)2 个不同的形式，(a)在发起站点上引用，其作用是在 tm 进程上为该事务建立 2 个消息端口：tm_p:tid 是与其伙伴站点上的 tm 进程通信的消息端口；rbm_p:tid 是与本站点上 rbm 进程通信的消息端口. (b)是在每个伙伴站点上引用的，其作用是为该事务在伙伴站点的 tm 进程上建立 2 个消息端口：tm_p:tid 是与事务发起站点上的 tm 进程通信的消息端口，rbm_p:tid 是与本站点上 rbm 进程通信的消息端口. 并且与事务发起站点上 tm 进程之间建立通信链路. 标记为“site_name/tm. tm_p:tid”的 socket 指示的是名为 site_name 的站点上 tm 进程的消息端口 tm_p:tid；标记为“filter(tm, tm. site_name)”的 socket 所表示的是名为 site_name 的站点上的 tm 进程.

(3)start_rbm(site_name, tid)

这一产生式的作用是在名为 site_name 的站点上为由参数 tid 所指定的事务创建一个 rbm 进程，并且在该进程与本站点的 tm 进程之间建立消息端口和通信链路. 在 rbm 进程上为本事务所创建的消息端口是 tm_p:tid，2 个 socket 中的嵌入表达式的值分别代表本站点上的 tm 进程和该 tm 进程上的消息端口 rbm_p:tid. 该产生式如图 7 所示.

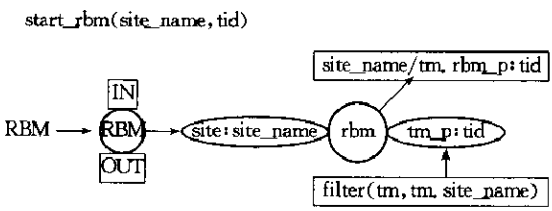


图7 产生式start_rbm()

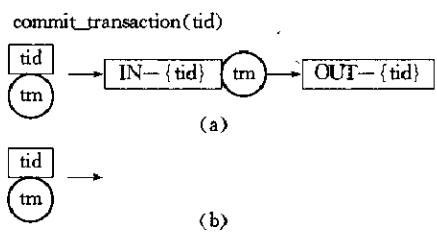


图8 产生式commit_transaction()

(4)commit_transaction(tid)

说明当事务(由参数 tid 指定)完成(提交)时，系统内进程结构及进程通信关系的变化. 当事务提交时，要撤消该事务所对应的 rbm 进程(由(b)说明)，取消 tm 进程上与该事务有关的所有消息端口和通信链路(由(a)说明). 该产生式如图 8 所示.

附录 A 给出了 sample 系统的 CSDL 语言描述，这是一个经过简化的例子，其中综合说明了以上各项内容以及各个图文法产生式的触发条件等. 将这样的 CSDL 语言描述交给 GRADECS 系统处理之后，GRADECS 系统会将其转换成运行结构和通信结构的内部表示，同时自动为 sample 系统生成相应的动态进程控制机制和进程通信支持机制. 根据 sample 系统的 CSDL 描述，GRADECS 负责在 sample 系统运行时自动地建立和维护其动态进程结构和动态进程通信. 图 9 所示的是 sample 系统某一运行时刻的系统状态图，为了表述简洁，其中略去了 Axiom 图的内容. 图中有 3 个站点：a，b 和 c，站点 a 和 b 在合作加工事务 ta1，站点 b 和 c 在合作加工 ta3. 在图中清楚地给出了这一时刻 sample 系统的进程结构、各

进程的消息端口以及各进程之间的通信关系和合作关系.图 9 实际上是由 sample 系统的 Axiom 图,通过引用相应的图文法产生式进行若干次重写所得到的状态图.

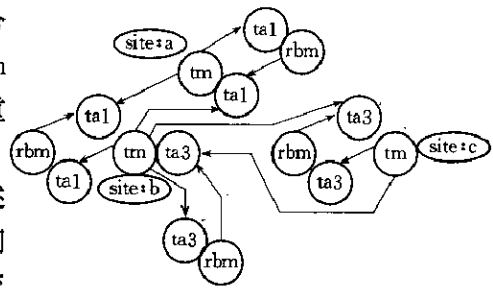


图9 sample系统某一运行时刻的状态图

这个例子除了表明用图文法可以很好地描述并发系统的结构和支持动态进程通信,还给我们这样的启示:运用这套图文法,能够得到并发系统运行时任一时刻的系统状态.这一系统状态包括:并发系统当时的进程结构、所有的消息端口、各进程之间的通信链路等重要信息,这些信息对并发系统的测试或调试过程是非常难得的.因此,在 GRADECS 系统中还以图文法为基础提供了一套并发系统的测试手段.在后面的章节中要具体介绍这些功能及其实现方法.

3　并发系统运行状态的测试

3.1　并发系统状态测试中的问题

在并发系统的调试运行期间,人们迫切需要了解其详细的系统运行状态,看其是否符合系统的规范要求.并发系统的运行状态包括:系统的进程结构、进程间的通信关系、各消息端口的状况与消息通信记录等.

目前已经有不少并发程序的 debugger[6],它们在多进程或分布式的环境下具有以下几项主要功能:重现执行过程(Execution Replay);设置检查点和回退(Checkpointing and Roll Back);设置断点(Breakpointing);目标程序挂起(Halting the Target Program).

尽管许多这样的 debugger 已具有多进程的能力,但无法跟踪和表示多进程之间,特别是跨站点的多进程之间的动态的通信和同步关系,对多进程并发系统运行期间的随意性无能为力.比如,对一个进程设置断点后,往往会改变整个系统的行为.

在并发系统的开发环境中,调试与查错工具的重点应放在对进程状态的监视、对进程通信的跟踪以及如何准确及时地提供并发系统的全局状态这几方面.通过这几方面来了解并发系统的详细运行状态,看是否符合其规范说明(Specification).由于我们采用图文法来作并发系统的 specification,使得我们能够从完整系统的角度给出一套以图文法为基础的并发系统状态测试手段.

3.2　GRADECS 系统中并发系统状态测试子系统的功能和组成

在 GRADECS 系统中,基于图文法的并发系统描述手段与状态测试手段很好地结合在一起.图 10 给出的是 GRADECS 系统的组成,其中的运行期状态测试控制器、运行期状态记录器以及为这二者提供数据服务的分布式数据库系统一起构成了 GRADECS 系统的并发系统状态测试子系统.该部分的基本功能是根据并发系统开发者的测试要求,及时准确地记录和整理并发系统运行状态和进程间消息通信的情况,为开发者提供全局性的系统状态图以及各消息端口、各功能簇上的消息通信纪录等系统调试期间所必需的信息.

GRADECS 提供了并发系统的测试要求说明语言 TDL(testing description language),并发系统开发者用 TDL 语言来说明自己的测试要求、测试结果的输出时机和输出形式.用 TDL 语言描述的并发系统测试要求经过并发系统状态测试控制器解释后,将其中对并发系

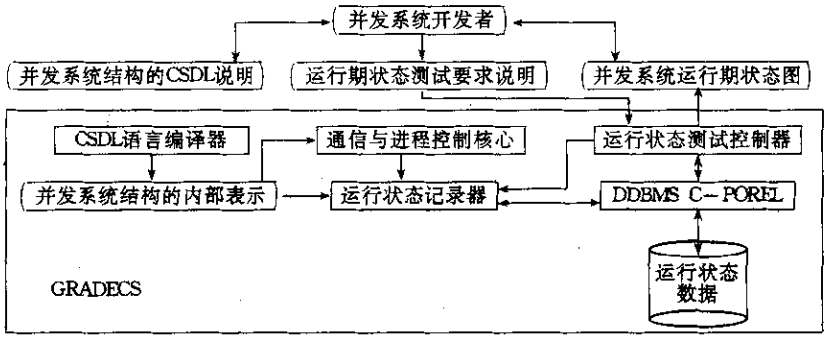


图10 GRADECS系统的结构

统运行期的监视记录要求交由运行期状态记录器处理,将测试纪录数据的输出要求留给自己处理.

由运行期状态记录器记录下来的状态数据量比较大,而且这些数据又分别来自并发系统的各个不同站点,因此,要选用一个全局的数据库管理系统来管理和存储这些数据.在实现时我们选用了分布式数据库管理系统C—POREL,利用它来存储和管理来自不同站点的测试数据,可以保证在系统内的任一站点上都能随时提取并发系统的全局性的状态测试数据,为并发系统状态测试控制器能向开发者提供全局状态图提供了方便可靠的支持.

3.3 并发系统的测试要求说明语言TDL

TDL中包括2大部分内容:①指明要记录哪些系统运行状态数据;②指明这些数据的输出形式和输出时机.前者称为记录说明,后者称为输出说明.

记录说明有以下的内容:

说明是否跟踪记录某一并发系统的动态运行结构,即记录该系统完整的系统状态图及其变化过程,直至系统运行结束或某一条件满足时为止;说明是否跟踪记录某一消息端口或某一类消息端口上的全部或指定类型的消息;说明是否跟踪记录某一或某一类进程(某个功能簇)的消息通信;说明是否跟踪记录某些站点上的系统运行状态,即并发系统的某个局部状态.

输出说明中有下面的内容:

测试数据的输出形式——图形或表格数据;测试数据的输出时机,其中包括:(1)并发系统运行完毕后输出;(2)引用某一产生式后输出;(3)收到或发出某一类消息时输出;(4)并发系统中某个事件发生时输出;测试数据的输出地点——指定在哪个站点的哪台设备上输出.附录B中给出了TDL语言的定义.

3.4 并发系统运行状态的测试过程

用CSDL语言进行系统结构说明的并发系统,可以用TDL语言对其作测试要求说明.GRADECS的状态测试子系统将TDL语言测试要求说明翻译成系统内部表示,对TDL语言的翻译主要做下面3项工作:(1)在并发系统的内部表示中对所有要求测试的状态数据结构打上测试记录的标记;(2)对于以表格形式输出的测试数据,根据TDL语言中的输出模式定义,为其在分布式数据库中定义输出数据的关系模式;(3)建立一张记录事件表,将TDL语言中记录条件说明部分所定义的测试数据记录时机与记录条件登记在该表中,以供运行期状态记录使用.

并发系统在调试运行时，GRADECS 系统会根据用户的测试要求，随时记录并输出并发系统的状态图或测试数据. GRADECS 系统的并发系统状态测试子系统对系统状态的测试及测试数据的记录与输出，是与并发系统的运行同时进行的，不去中断任何进程，不会对并发系统的运行状态产生影响. 并发系统开发者可以在其并发系统运行的同时，看到较为实时的系统运行状态信息和测试数据，也可以在其并发系统运行结束后，一步步地对系统运行状态进行追踪，以发现运行过程中的问题. 当发现某个进程出现问题时，可对该进程进行隔离，然后采用传统的单进程 debugger 对其进行内部查错处理. 在下一小节中，我们给出几个 TDL 语言的应用例子.

3.5 并发系统运行状态测试的例子

这里我们仍以上一章中的 sample 系统为例. 对 sample 系统来讲，其测试重点在于检查各个功能簇在处理分布式事务时的配合情况以及各进程之间的通信情况.

例 1：当一个新事务启动时，检查各伙伴站点的状态.

```
SYSTEM sample
TEST_BEGIN
  RECORD STATES ON site_x
    WHEN SEND MSG OF TYPE 'new_transaction' TO site_x,
  OUTPUT GRAPH ON terminal_1
    AFTER SEND MSG OF TYPE 'new_transaction';
TEST_END.
```

这段 TDL 测试说明的作用是：在启动一个新事务时，要求系统记录下伙伴站点 site_x 上的进程状态，并且在启动消息发出后将该伙伴站点的系统状态图输出到终端 terminal_1 上(在多窗口环境下，terminal_1可以指示某个窗口). 设 site_x 为站点 b，其输出可如图11所示. 其中站点 b 正与站点 a 合作加工事务 ta1，与站点 c 合作加工事务 ta3.

例2：查看某一站点上 tm 和 rbm 之间的通信和配合情况.

```
SYSTEM sample
TEST_BEGIN
  RECORD MESSAGES OF site_a. rbm FROM site_a. tm,
  OUTPUT TABLE (MSG_NO, MSG_TYPE, PORT, PID,
  MSG) ON terminal_1;
  RECORD MESSAGES OF site_a. tm FROM site_a. rbm,
  OUTPUT TABLE(MSG_NO, MSG_TYPE, PORT, PID, MSG) ON terminal_1;
TEST_END.
```

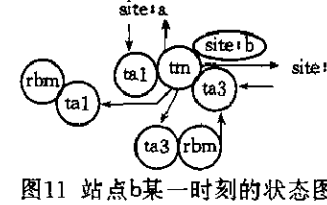


图11 站点b某一时刻的状态图

这一 TDL 说明的作用是：当 sample 系统运行结束后，让系统输出形式如图12所示的表格型数据. 同样也可以记录下 rbm，tm 两个功能簇分别发往对方的消息：

```
RECORD MESSAGE OF site_a. rbm TO site_a. tm,
OUTPUT TABLE(...) ON terminal_1;
RECORD MESSAGE OF site_a. tm TO site_a. rbm,
OUTPUT TABLE(...) ON terminal_1;
```

例3：当某一站点发生故障或从故障中恢复时，输出当时的全局运行状态，以检查故障对并发系统运行的影响或故障站点的恢复工作如何. 此时的测试要求可说明如下：

```
SYSTEM sample
TEST_BEGIN
  RECORD ALL STATES WHEN SITE CRASHED,
  OUTPUT GRAPH AFTER SITE CRASHED;
  RECORD ALL STATES WHEN SITE RECOVERED,
```

OUTPUT GRAPH AFTER SITE RECOVERD;
TEST_END.

site_a.rbm:

MSG_NO	MSG_TYPE	PORT	PID	MSG
1	start	ta1	rbm1	…
3	data	ta1	rbm1	…
4	data	ta1	rbm1	…
7	start	ta3	rbm2	…
10	data	ta3	rbm2	…
12	commit	ta1	rbm1	…
13	commit	ta3	rbm2	…

site_a.tm:

MSG_NO	MSG_TYPE	PORT	PID	MSG
2	wait	ta1	rbm1	…
5	ready	ta1	rbm1	…
6	wait	ta3	rbm2	…
8	ready	ta3	rbm2	…
9	commited	ta1	rbm1	…
11	abort	ta3	rbm2	…

图12 消息记录表格

图13是站点site_a故障后的系统状态图,它表明事务ta1受到站点故障的影响而无法执行下去.图14是站点site_a恢复后的系统状态图,它表明事务ta1得到恢复,并以新的事务标识ta1′重新执行.

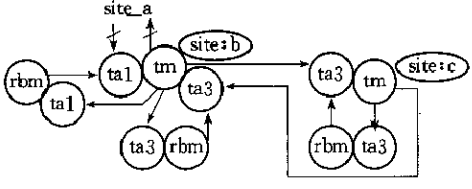


图13 站点a故障后的系统状态图

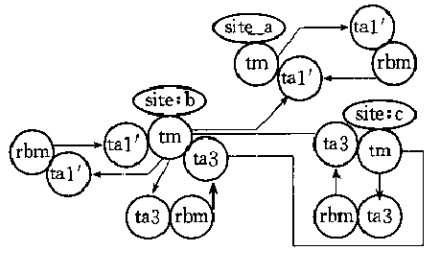


图14 站点a恢复后的系统状态图

4 GRADECS系统中并发系统状态测试子系统的实现算法

在并发系统状态测试子系统中有4个主要算法:TDL语言的编译、测试数据管理(包括测试数据库模式的建立和对测试数据库的存取)、测试数据记录条件的监测、运行状态图推演器算法.其中最核心的算法是状态图推演器算法.这一章将详细讨论此算法.

4.1 并发系统运行状态图推演器的作用

并发系统开发者在TDL测试说明中,会要求随时能输出其并发系统的运行状态图.如果只要求在其并发系统运行结束后重演状态图的变化过程,便可采取在并发系统运行时只记录每次图文法产生式的具体调用(包括:产生式名、具体调用参数值和调用者的说明)的方法.在并发系统运行结束后,可根据记录下来的图产生式调用序列,从其Axiom图出发,一

步步推演出并发系统此次的全部运行状态.在具有全局时钟的系统中,这种要求容易实现.但是,如果并发系统开发者要求在其并发系统运行的同时,随时能看到并发系统最近时刻的运行状态图,问题就变得很复杂了.首先,为了系统的健壮性,即在某些站点或通信链路出现故障时,仍能向开发者提供并发系统的运行状态图(这一点往往是在并发系统调试过程中最需要的),必须将与并发系统实际运行相同步的状态图推演在各个站点上同时进行,在需要时可从任何一个站点上输出.其次,由前面所讲的同步推演带来了复杂的各站点上的推演过程和推演出的状态图的一致性问题.

为解决上述问题,我们根据 Lamport 的时间行为概念[7],设计并实现了一个基于全局逻辑时钟的具有高度容错性的并发系统同步算法——时间状态机算法(Time State Machine Algorithm),它是 GRADECS 中并发系统状态图推演器的核心.

4.2 时间状态机算法

凡是涉及到通信和时间的系统中,超时(Timeout)是一个最基本和最有用的概念,它表明的是一种关于时间的约定.超时一般用来检测系统的故障.超时的使用基于对系统的时间行为的假设,如:设置最长消息通信延迟时间,或进程完成某一动作的最长时间等.当这些关于系统时间行为的假设中所设置的时限被超过时,便可推知系统内产生了异常.通过对系统时间行为的假设是否还能感知更多的信息,而不仅仅是系统异常呢?Lamport[7]曾经讨论过使用系统时间行为假设来构造高容错度的分布式系统的问题.下面我们通过对 GRADECS 系统及其实现环境的时间行为假设(称为一组前提),来推出 GRADECS 系统的时间状态机算法.

4.2.1 系统时间行为

在 GRADECS 系统中,状态图推演器在每个站点上都对应一个独立进程.当某一条图文法产生式被引用时,GRADECS 系统便通过其通信支持机制,将一条产生式引用消息广播至各个站点上的状态图推演器进程.由此,我们将所讨论的问题抽象成如下的模型:由通信链路互连在一起的进程网络,其中每个进程都执行事件驱动的算法.这里的事件指的是一条消息的到达或进程的时钟到了某一指定时刻.在这个模型中,超时的应用基于如下的假设:

(A1) 对任意一个使得进程 p_i 向进程 p_j 发送消息 m 的事件 e,存在一个常量 δ,使得:若 e 发生在时刻 T,且 p_i 和 p_j 及连接 p_i 与 p_j 的通信链路均正常,则消息 m 在时刻 $T+\delta$ 应到达进程 p_j.其中时刻 T 和消息到达时刻 $T+\delta$ 均由同一进程(p_i 或 p_j 之一)的时钟读出.

在(A1)中最重要的一点是:δ 是事先确定的通信延迟,它不依赖于哪一次具体的消息收/发,也不受系统内消息流量和资源使用情况的影响.延迟 δ 由下面2部分构成:处理事件 e 和生成消息 m 所用的最长时间的下限、通过通信链路传输消息 m 所用的最长时间的下限.

为了能给事件处理和消息生成时间定界,要求系统有足够的运算和处理能力来应付可能发生的大量突发事件;为了能给消息传输时间定界,又要求系统所使用的通信链路具有足够的带宽来容纳可能产生的最大消息流量.严格地讲,这2条要求是较难达到的.这时就要选择一个合理的 δ 值,既使得几乎所有事件所生成的消息都能在时间 δ 内到达,又得保证系统有较高的效率.但是当处理器过载或通信链路突然拥塞了大量消息时,仍会有某些消息在时间 δ 内到达不了目的地.此时,从假设(A1)的角度来看,便认为系统内发生了故障.在实际确定 δ 时,需要认真权衡,选择一个使此类故障的概率低于某一合理值的最小 δ 值.

为了以后叙述方便,我们假设每个进程可以向自己发消息,即(A1)在 $i=j$ 时仍然成立.

为了满足(A1),系统内所有非故障进程都要有自己的时钟,这些时钟的速率(或频率)基本相同.因此,我们有下面假设:

(A2) 任意2个非故障进程的时钟 c_i 和 c_j 在任意物理时刻 t,最多相差常量,即:

$$|c_i(t)-c_j(t)|<\varepsilon.$$

为了满足(A2),各进程的时钟必须周期性地进行同步.目前有许多在各种故障模型下的时钟同步算法[8]可以满足我们的要求,因此,可以放心地使用这一假设.从(A1)和(A2)可以推出下面的结论:

推论1. 设 e 是进程 p_i 中的事件,若事件 e 发生的时刻在进程 p_i 的时钟看来是时刻 T,且 e 使得进程 p_i 向进程 p_j 发送消息 m,并且 p_i 与 p_j 之间的通信链路正常,则:从 p_j 的时钟来看,到时刻 $T+\delta+\varepsilon$ 为止,消息 m 一定到达 p_j.

为了导出我们的算法,还需要下面这条假设:

(A3) 一个进程可以确定它所接收到的消息的直接源(Immediate Source).

在(A3)中,所谓直接源是指:若消息 m 从进程 p_i 经通信链路直接到达进程 p_j,途中不通过其他进程,则在 p_j 看来,p_i 就是 m 的直接源.(A3)并不要求 p_j 了解消息 m 的最初源(Original Source).

对于不同的故障模型,(A3)得到满足的难易程度差异极大.在故障停机处理器模型(Fail—Stop Processor Model)[9],只需要在消息上附上直接源的地址,但是对 Byzantine 恶意故障模型(Malicious Byzantine Failure Model)[10],则需要非常复杂的协议来保证.

在 GRADECS 系统中,我们通过对大量实验数据的分析及实现分布式数据库管理系统 C—POREL 的经验,确定了常量 δ. GRADECS 的通信支持机制中采用 C—POREL 的成功算法,提供了满足(A2)的全局逻辑时钟.[11]另外,GRADECS 的实现是基于 fail—stop processor model 的,因此保证(A3)得到了满足.

有了上面的3条假设和全局时钟,我们便可引入下面这一有趣的概念——靠不发消息来传递信息(Send a Message by Not Sending a Message).

4.2.2 如何实现不发消息而能传递信息

在全局时钟的支持下,一个进程就能不做任何实际动作而向其他进程传递信息.下面给出不做动作而发消息的算法,在这一算法中,所发送的消息被接受方解释为一条特殊的 NULL 消息.

算法1.

由进程 p_i 在 T 时刻向进程 p_j 发送带有时戳(Time Stamp)T 的消息 m.

```
process i:
    WHEN clock=T DO {
        IF m≠NULL THEN send message T:m to j;
    }
process j:
    WHEN clock=T+δ+ε DO {
        IF exactly one T:m message has been received from i
        THEN message received:=m;
        ELSE message received:=NULL;
    }
```

算法1中最后一个 IF 语句中的“exactly one”是用来处理当进程 p_i 由于故障而发送空消息的情况. 在算法1中，p_i 按一般方法发送非空消息. 当 p_i 发送带时戳 T 的空消息时，p_i 不用做任何动作. 在 fail－stop 故障模型中，进程在发生故障后便停止运行或被挂起，即一个故障进程只能对外发送空消息. 因此，空消息的含义之一便是报告故障.

对于算法1我们有如下结论：

推论2. 在算法1中，若进程 p_j 是正常的，则：

(*a*)在 p_j 的时钟到达时刻 $T+\delta+\epsilon$ 为止，p_j 能收到 p_i 发来的带有时戳 T 的消息 m；

(*b*)若 p_i 与 p_j 之间的通信链路正常，则 p_j 收到的消息即是 p_i 发出的消息.

推论2是推论1和(A3)的直接推论.

使用算法1我们可以使进程 p_i 在每个时钟 *tick* 向 p_j 发送一条消息，如果每个 *tick* 是一个毫微秒(*ns*)，那么每一秒钟便会有10^9条消息发出. 当然这里绝大多数是空消息，不会增加系统的通信负担. 这些空消息在逻辑上的存在是我们构造后面的通用并发同步算法的基础.

在并发系统中，需要互相同步的各个进程不仅需要两两之间的可靠消息通信，而且需要可靠的广播消息通信，使一个进程能同时与其他所有进程或某一组进程通信. 我们假设存在这样一个由下层通信网络的基本 message passing 操作构成的可靠消息广播算法[12]，而且该算法满足下面的实现条件，其中△为某个常量：*IC*.

若进程 p_i 在其时钟为 T 的时刻向外广播消息 m，P 为 m 的目的进程集合，则存在常量△(△$\geqslant\delta$)，使得：

(a)若进程 p_i 正常，则任意的一个正常进程 p_j($p_j\in P$)在其时钟到达时刻 $T+$△时为止，定能收到消息 m；

(b)若 p_j 与 p_k 为 P 中的任意2个正常进程，则它们在各自的时钟到达时刻 $T+$△为止，或者都收到 p_i 发出的同一条消息 m，或者都没有收到消息 m.

4.2.3　时间状态机算法

在这一部分中我们就给出一个通用的多进程并发同步的算法. 由于这个算法的基础是系统的时间行为，因此称为时间状态机算法.

算法的基本思想是：设有 N 个进程，在每个时刻 T，每个进程 p_i 都向所有进程广播一条命令 $C_{i,T}$($C_{i,T}$可以是空命令)，在时刻 $T+$△，各进程发出的命令 $C_{1,T},C_{2,T},\ldots,C_{N,T}$将按某种一致的次序在所有进程上执行. 任何事先设定的超时处理动作将通过执行一条特殊的 timeaction 来施行. 所谓 timeaction 就是：当所预期的到某一时刻应该发生的事件在该时刻到达时并未发生，系统所应采取的处理措施. 算法中的每一条命令可看作一个例程(Subroutine)，这样的例程具有3个参数：状态、时间和例程标识(timeaction 除外). 其中的状态参数是按“callby reference”方式引用的. 执行某一命令例程时，会改变其状态参数的值，新的状态参数将被带入下一命令例程的执行. 状态参数的这种传递，可看作是进程自己给自己发消息. 至于状态参数的值如何改变，则有赖于具体命令例程的内部算法了.

设：任意一个进程 p_i 的起始时钟读数为 *clock*，起始状态为 *state*，则各命令的生成与执行将按下面的算法进行：

算法2.

p_i: LOOP FOREVER

```
FOR j=1 TO N DO
  execute C_{j,clock-△}[state, clock-△, j];                    (1)
execute timeaction[state,clock-△];                             (2)
generate command C_{i,clock} and
broadcast C_{i,clock} to all other process;                    (3)
clock:=clock+1;                                                (4)
END LOOP
```

算法2中的语句(4)是靠时钟的自动增值实现的. 设 $T=clock-\triangle$,语句(1)和(2)构成了状态机的 T 时刻执行步. T 执行步的起始状态为 $state_{T-1}$,T 执行步完成后的状态(结束状态)为 $state_T$. 每一进程 p 当其时钟到达时刻 $T+\triangle$时,都执行以下步骤:(a)执行状态机的 T 时刻执行步,并计算出 $state_T$;(b)广播自己的 $T+\triangle$时刻的命令.

对于算法2,文献[7]证明了如下的结论:

推论3. 若广播通信机制满足条件 IC,则算法2在任意时刻 T 满足下列要求,其中 $\triangle'=\triangle+\delta$:

(a)对任意进程 p_i,若 p_i 在其时钟到达时刻 $T+\triangle'$为止,未曾发生故障,则:p_i 在到达 $T+\triangle'$时刻为止已经执行了状态机的 T 时刻执行步,并且在执行时使用了它在 T 时刻所生成的命令 $C_{i,T}$;

(b)任意2个到 $T+\triangle'$时刻状态仍然正常的进程,直到 T 时刻执行步,它们所执行的是相同的状态机执行步序列.

由推论3可知,算法2通过使各个进程执行一致的命令序列,确保了所有进程的同步. 因此,在 GRADECS 系统中,我们采用这一算法在各个站点上推导出一致的并发系统运行状态图.

4.2.4 时间状态机算法在 GRADECS 系统环境下的实现要求

为了保证各个状态图推演器进程在同时推导并发系统运行状态图时的相互同步,以推出一致的状态图,每个站点上的状态图推演器进程都执行与算法2类似的算法. 算法2中的 $C_{i,T}$具体化为图文法产生式引用命令,状态参数 $state$ 的值便是具体的每一步所推出的系统状态图. 根据推论3可知,每个状态图推演器进程所执行的是相同的图文法产生式引用命令序列,因此,保证了各站点独立导出的并发系统状态图之间的一致性. 但是,是否能保证所导出的状态图的正确性,即状态图是否准确地表达了并发系统的实际状态变迁,还需进一步证明.

保证所导出的并发系统状态图的正确性,就是要保证各推演器进程所执行的产生式引用命令序列与并发系统运行中实际的产生式引用序列相一致. 在并发系统中,决定系统状态变化的主要因素是并发系统中的不同事件之间的"引发"关系(Causual or Happened—Before Relation)[12],简记为关系"→". 关系"→"是一个偏序关系. 在同一进程中,若事件 e_i 发生在 e_j 之前,则有:$e_i \rightarrow e_j$;若事件 e_i 和 e_j 分属于不同的进程,并且 e_i 发出消息 m,而 e_j 接收消息 m,则:$e_i \rightarrow e_j$. 各状态图推演器进程在执行算法2时,应该保持各产生式引用之间的关系"→". Lamport 在文献[2]中给出了一个保持系统中各事件间的"→"关系的全局逻辑时钟. 我们在高速通信介质和可靠广播通信的支持下,将 Lamport 的全局逻辑时钟改造成了既保持"→"关系又满足(A2)的全局时钟.[11]对这一全局时钟系统有下面的假设:

(A4) 一个时钟系统 $CLOCK=\{c_i \mid i=1,2,\ldots,N\}$,其中 c_i 是进程 p_i 的(或 p_i 能读到

的)时钟，N 是进程数，$c_i(t)$为物理时刻 t 时 c_i 的读数. $CLOCK$ 满足：对于任意物理时刻 $t1$ 和 $t2$，若 $C_{i,t1}\rightarrow C_{j,t2}$，则有 $c_i(t1)<c_j(t2)$.

对于任意2次图文法产生式引用 C_i 和 C_j，如果 C_j 引用的产生式的左部直接或间接地是 i 的结果(可以说 C_i 有消息传递给 C_j)，这时便有 $C_i\rightarrow C_j$；否则，C_i 与 C_j 之间不存在"→"关系. 关于图文法产生式的引用序列，有如下的假设：

(A5) 任意2次图文法产生式引用 C_i 和 C_j，若 $C_i\rightarrow C_j$，则在产生式引用序列中有 C_i 在 C_j 之前；若 $C_i\not\rightarrow C_j$ 且 $C_j\not\rightarrow C_i$，则产生式引用序列$\{\ldots,C_i,\ldots,C_j,\ldots\}$与 C_i 和 C_j 互换位置后的序列$\{\ldots,C_j,\ldots,C_i,\ldots\}$的结果等价.

下面给出关于算法2的定理：

定理. 算法2能保持图文法产生式引用的正确顺序.

证明：设 $C_{i,T1}$和 $C_{j,T2}$为任意2条产生式引用命令，并且 $C_{i,T1}\rightarrow C_{j,T2}$. 由假设(A4)可知：$T1<T2$，即 $C_{i,T1}$所处的 $T1$时刻执行步先于 $C_{j,T2}$所处的 $T2$时刻执行步. 所以，算法2保持了各图文法产生式引用之间的"→"关系.

设 $C_{i,T1}$和 $C_{j,T2}$之间不存在关系"→". 若 $T1<T2$，则 $C_{i,T1}$所处的 $T1$时刻执行步先于 $C_{j,T2}$所处的 $T2$时刻执行，即2次产生式引用是按全局时钟给出的时间顺序执行的；若 $T1=T2=T$，则二者同处于状态机的 T 时刻执行步，根据(A5)，在同一执行步内 $C_{i,T}$和 $C_{j,T}$的任意顺序都是等价的. 所以，算法2保持了并发系统中图文法产生式的正确顺序. □

4.3 算法实现上的考虑

时间状态机算法主要新在概念上，无论在理论形式上还是在实现上都比较简捷. 其核心是维护一个访问速度较高的命令队列，将所收到的命令按所附的时戳的顺序入队，并在 T 时刻将 $T-\triangle$时刻的命令出队执行. 在由高速消息通信互连机制支持的分布式存储多机体系结构中，可取$\triangle=\delta+\varepsilon$.

算法2是一个各进程完全自主、高度容错的多进程同步算法. 但这种高度自主、容错和同步的代价也是明显的，其中最显著的一点是降低了所导出的并发系统运行状态图的实时性，即状态图推演进程输出的状态图落后于并发系统当前状态一个$\triangle'$时间. 在由高速全互连通信机制支持的分布式存储多机体系结构中，有$\triangle=\delta+\varepsilon$ 成立，这时推出：$\triangle'=2\delta+\varepsilon$. 在我们的实现中，$\triangle'<60ms$，这也可以满足并发系统调试过程中一般的实时要求了；第2个代价是增加了系统内的消息流量. 因为这一算法的支持环境复杂(如：要有可靠广播通信和全局时钟的支持)，因此，增加了消息流量. 所以，该算法只适用于由高速宽带互连通信机制支持的分布式存储多机体系结构中.

5 结束语

并发系统的测试就是看并发系统在运行时是否符合其规范说明(Specification)，因此，并发系统的测试方法与规范说明方法应该有机地结合在一起. 在 GRADECS 系统中，我们尝试以图文法这一形式理论为基础，提供统一的、重点在并发系统的动态运行结构的并发系统形式化说明方法和运行状态测试方法. 由于基于图文法的并发系统状态测试方法与 GRADECS 所支持的并发系统结构说明、运行结构支持机制等，都是构造在统一的图系统之

内的,因此,它与并发系统的运行过程衔接的非常紧密和自然.在并发系统的运行调试阶段,并发系统开发者可以通过 TDL 语言来说明对并发系统的测试要求,如:规定测试点,说明要记录的测试数据类型、形式及输出时机等.在不影响并发系统运行状态的前提下,能够及时地得到并发系统的全局或特定的局部运行状态.

在这套测试方法的实现上,根据 Lamport 的时间同步概念,设计并实现了用于多站点进程并行推演并发系统运行状态图的时间状态机算法.该算法保证了并发系统全局状态的及时取得.该算法对多机系统的性能,特别是通信支持机制的性能要求较高(unix 操作系统,32位以上的处理器,10Mbps 以上的通信速率),但这种要求目前已不难达到了.

参考文献

1 Schneider F B, Andrews G R. Concepts for concurrent programming. LNCS 224, Springer—Verlag, 1985. 669~716.

2 徐建礼.基于图文法的并发系统开发支撑环境[博士论文].中国科学院数学研究所,1991.

3 徐建礼,周龙骧.用于描述面向对象并发系统的实用图文法模型.软件学报,1995,**6**(增刊):170~181.

4 Janssens D, Rozenberg G. On the structure of node—label controlled graph languages. Information Science, 1980, **20**:191~216.

5 Zhou L. C—POREL: a distributed relational database management system on microcomputer network. SCIENTIA SINICA(A), 1986,**29**(**1**):78~91.

6 Workshop on parallel and distributed debugging. Wisconsin, ACM SIGPLAN NOTICES, May 1989,**24**(**1**).

7 Lamport L. Using time instead of timeout for fault—tolerant distributed systems. ACM Transactions on Programming Languages and Systems, 1984,**6**(**2**):254~280.

8 Simons B, Lundelius J, Lynch N. An overview of clock synchronization. Technical Report, IBM Research Division, Oct. 1988.

9 Fischer M J, Lynch N A, Paterson M S. Impossibility of distributed consensus with one faulty process. Tech. Rep. MIT/LCS/TR—282, M. I. T., Cambrigde, Mass., Sept. 1982.

10 Lamport L *et al*. The byzantine generals problems. ACM Transactions on Programming Languages and Systems, 1982,**4**(**3**):382~401.

11 Xu J, Zhou L. CS: the communication subsystem of C—POREL. Advances in Chinese Computer Science, World Scientific, Singapore, 1991,**3**:110~131.

12 Lamport L. Time, clock and the ordering of events in a distributed system. CACM, 1978,**21**(**7**):558~565.

附录 A Sample 系统的 CSDL 描述

```
SYSTEM sample                          /* 系统名 */
  STATIC                               /* 系统静态结构说明 */
    MODULE TM:
      SITE all,
      PARAMETER (site_name, site_list);
    MODULE RBM
      SITE all,
      PARAMETER (site_name, tid);      /* 以上是对各系统模块的定义,其中包括:模块名 TM 和 RBM,各
                                          模块所在的站点名和各模块的启动参数表 */
    AXIOM                              /* 对系统 Axiom 的描述 */
      MODULE TM:
        PORT(tm_p, rbm_p)
        CONNECT_TO(RBM, tm_p)(TM, tm_p);
```

```
    MODULE RBM;
      PORT(tm_p)
      CONNECT_TO(TM, rbm_p);        /*说明每个模块上的端口以及该模块与其他模块的哪些端口相连
                                    */
    AXIOM_END;
  STATIC_END;

  DYNAMIC                           /* 定义系统的动态结构(定义系统中2个 f—cluster 的结构) */
    F_CLUSTER tm                    /* 对事务管理 f—cluster 的说明 */
    F_CLUSTER_BEGIN
      MODULE TM;                    /* 该 f—cluster 对应的程序模块名 */
      ON ALL SITE;                  /* 该 f—cluster 的进程分布在系统内所有站点上 */
      PRODUCTION tm_start(site_name),
                 start_transaction(site_name, site_list, tid),
                 commit_transaction(tid);
                                    /* 该 f—cluster 共使用3条图文法产生式 */
      TRIGGER                       /* 对产生式引用条件的说明 */
        EVENT system_init: tm_start;  /* 在系统初启时引用 tm—start */
        ON MESSAGE TYPE 'new_transaction': start_transaction;
                                    /* 在收到类型为'new_transaction'的消息时,引用 start_transac-
                                       tion */
        ON MESSAGE TYPE 'commit_transaction': commit_transaction;
                                    /* 在收到类型为'commit_transaction'的消息时引用产生式 com-
                                       mit_transaction */
      TRIGGER_END;
    F_CLUSTER_END;

    F_CLUSTER rbm                   /* 对数据操作 f—cluster 的说明 */
    F_CLUSTER_BEGIN
      MODULE rbm;
      ON SITE ALL;
      PRODUCTION rbm_start(site_name, tid),
                 commit_transaction(tid);
      TRIGGER
        ON MESSAGE TYPE 'do_it':rbm_start;
        ON MESSAGE TYPE 'commit': commit_transaction;
      TRIGGER_END;
    F_CLUSTER_END;
  DYNAMIC_END;
SYS_END.
```

附录 B　TDL 语言定义

```
〈并发系统测试说明〉::=SYSTEM〈并发系统名〉
                    〈测试说明〉
〈测试说明〉::=TEST_BEGIN
                  {〈测试数据记录说明〉〈测试数据输出说明〉;}
              TEST_END.
〈测试数据记录说明〉::=RECORD〈记录数据类型说明〉[〈记录条件说明〉]
〈记录数据类型说明〉::=ALL STATES
                      |STATES ON〈站点名表〉
                      |MESSAGES[TYPE〈消息类型表〉]
                      ON PORT(ALL|〈PORT 名表〉)
                      |MESSAGES OF〈f—cluster 名表〉
                      [FROM〈f—cluster 名表〉]
〈记录条件说明〉::=〈系统条件表达式〉
〈系统条件表达式〉::=〈系统条件〉
                   |'('〈系统条件表达式〉')'
```

|〈系统条件表达式〉〈逻辑运算符〉〈系统条件表达式〉
〈系统条件〉::=〈产生式引用条件〉
|〈收/发消息类型条件〉
|〈超时条件〉
|〈站点故障条件〉|〈站点恢复条件〉
|〈事件条件〉
〈产生式引用条件〉::=[BEFORE|AFTER]PRODUCTION 〈产生式名表〉INVOKED
〈收/发消息类型条件〉::=(WHEN|AFTER)[RECEIVE|SEND]MSG OF TYPE
〈消息类型表〉[(FROM|TO)〈站点名表〉]
〈超时条件〉::=WHEN TIMEOUT ON 〈时计名表〉
〈站点故障条件〉::=WHEN SITE[〈站点名表〉]CRASHED
〈站点恢复条件〉::=WHEN SITE[〈站点名表〉]RECOVERED
〈事件条件〉::=WHEN EVENT 〈用户定义事件表〉 HOLD
〈测试数据输出说明〉::=OUTPUT 〈输出方式说明〉[〈输出时机说明〉]
〈输出方式说明〉::=〈图形输出说明〉|〈表格输出说明〉
〈图形输出说明〉::=GRAPH[ON 〈图形终端号〉]
〈表格输出说明〉::=TABLE 〈表格定义〉[ON 〈终端号〉]
〈输出时机说明〉::=〈系统条件表达式〉
〈表格定义〉::='('〈域名〉{,〈域名〉')'
〈域名〉::=〈消息序号栏〉|〈消息类型栏〉|〈端口名栏〉|〈进程标识栏〉
|〈f—cluster 标识栏〉|〈产生式名栏〉|〈站点名栏〉
〈消息序号栏〉::=MSG_NO
〈消息类型栏〉::=MSG_TYPE
〈端口名栏〉::=PORT
〈进程标识栏〉::=PID
〈f—cluster 标识栏〉::=F_ID
〈产生式名栏〉::=PRODUCTION
〈站点名栏〉::=SITE
〈站点名表〉::='('〈站点名〉{,〈站点名〉')'
〈消息类型表〉::='('〈消息类型标识〉{,〈消息类型标识〉')'
〈PORT 名表〉::='('〈PORT 名〉{,〈PORT 名〉')'
〈产生式名表〉::='('〈产生式名〉{,〈产生式名〉')'
〈用户定义事件名表〉::='('〈用户定义事件名〉{,〈用户定义事件名〉')'
〈时计名表〉::='('〈时计名〉{,〈时计名〉')'
〈站点名〉::=string|number
〈消息类型标识〉::=string
〈PORT 名〉::=string
〈产生式名〉::=string
〈用户定义事件名〉::=string
〈时计名〉::=string
* 注:string 为定长字符串,number 为正整数.

A GRAPH GRAMMAR BASED CONCURRENT SYSTEM TESTING METHOD AND ITS IMPLEMENTATION

Xu Jianli Zhou Longxiang

(*Institute of Mathematics The Chinese Academy of Sciences Beijing* 100080)

Abstract In the research and development of concurrent systems, there is an urgent need for a formal method which can effectively specify the dynamic process topologies and dy-

namic inter—process communication behaviors of concurrent systems, and for a system testing method which is based on the formal specification method and can detect the dynamic transformations of system states. This paper gives a system testing method of concurrent systems which is based on graph grammar. The testing method can automatically trace and record the running states and communication histories according to the testing requirements given by system developers. The developers can get the current system state of a running concurrent system in realtime, or replay the transformation process of system states at any timeneeded. The implementation algorithms of this method are also described.

Key words Concurrent system, system testing, graph grammar, formal method, fault—tolerant and synchronization algorithm.