基于有限状态进程的事件约束定义*

顾 庆, 陈道蓄, 谢 立, 韩 杰, 孙钟秀

(南京大学 计算机软件新技术国家重点实验室,江苏 南京 210093)
E-mail: guq@dislab.nju.edu.cn
http://www.nju.edu.cn

摘要: 测试分布式程序需要定义事件约束来检测程序执行产生的事件序列.事件约束需要根据程序的规约来推导.FSP 是一类描述并发程序形式化规约的进程代数记法.它将并发进程描述为动作序列,其中动作可对应到规约级事件.E-CSPE 约束在给定状态谓词下定义前后运行事件间的顺序关系.根据 FSP 的操作符和并发控制机制可推导 E-CSPE 约束.推导出来的 E-CSPE 约束考虑到并发程序的安全和进展属性,可据以判断程序运行的正确性和测试的充分性.

关 键 词: 软件测试;有限状态进程;基于规约的测试;并发程序;事件约束
中图法分类号: TP311 文献标识码: A

由于并发的存在,分布式程序的执行具有不可预测性和不可重现性.这使得分布式程序的测试工作变得十分困难.一种可行的方案是将分布式程序的执行看作一个同步动作序列,在程序执行时将动作序列记录为一个可行同步事件序列(feasible SYN-sequence),根据该序列可以检测程序执行是否存在问题.

检测事件序列时需要描述什么是有效的事件序列(valid SYN-sequence),这要求有一个事件约束的描述规则.CSPE[1](constraints on succeeding and preceding events)是一个较为完善的事件约束描述方案.它规定了前、后事件间的依赖关系以及这种关系的或然性.CSPE 在描述事件约束时存在一个较大的缺陷,即它忽略了程序的运行状态,而事件间的依赖关系极有可能取决于事件发生时程序所处的状态.

E-CSPE(extended CSPE)[2]是 CSPE 的一种有效扩充,它将事件约束置于程序状态谓词的控制之下,使得对前、后事件间依赖关系的描述更为准确,从而增加了程序测试的可靠性.但使用 E-CSPE 约束描述规则时存在一个问题,即如何从程序规约推导出足够数量和有效的 E-CSPE 事件约束描述——简称 E-CSPE 约束.本文基于并发程序的 FSP(finite state process)表示探讨如何根据程序形式化描述推导出有效的 E-CSPE 约束集.

1 并发程序的 FSP 表示

1.1 FSP简介

FSP(finite state processes)[3]是LTSA(labeled transition system analyzer)中采用的一类进程(process,是指并发程序的组成单元,FSP 中是递归的概念)记法.它可以描述一个并发程序,并验证其行为(behavior)与规约的一致性.FSP 的优点是可以直接从代数记法转换成对应的 Java 程序.在 FSP 中,进程按递归的形式定义并由有限状态机表示,对应的图形描述为 LTS(labeled transition system).

* 收稿日期: 2001-03-06; 修改日期: 2001-08-01
基金项目: 国家"九五"重点科技攻关项目(98-780-01-07-03)
作者简介: 顾庆(1972－),男,江苏常州人,博士,讲师,主要研究领域为分布式语言和系统;陈道蓄(1947－),男,江苏南京人,教授,博士生导师,主要研究领域为分布式计算,并行处理;谢立(1942－),男,江苏常熟人,教授,博士生导师,主要研究领域为分布式计算,并行处理;韩杰(1976－),男,江苏南京人,硕士,主要研究领域为分布式语言和系统;孙钟秀(1936－),男,江苏苏州人,教授,博士生导师,中国科学院院士,主要研究领域为分布式计算,并行处理.

在 FSP 中主要有以下一些操作符:

(1) 顺序:→,表示动作间的顺序.如 $x→P$ 表示先执行动作 x,再执行进程 P 描述的动作序列.

(2) 选择:|,表示不同执行动作的选择.如 $x→P|y→Q$ 表示或者先执行动作 x,再按照 P 的描述执行;或者先执行动作 y,再按照 Q 的描述执行.

(3) 条件控制:when,表示在给定条件下执行某一动作.如 when B $x→P|y→Q$ 表示:若条件 B 成立,则除了 $y→Q$ 以外可以选择执行动作 x 再执行进程 P 的动作序列;若 B 不成立,则只能执行动作 y,然后再执行进程 Q 的动作序列.这里,B 是一个状态谓词.

(4) 进程前缀(process prefixing):又称进程标注(process labeling),声明进程的一个实例.如"*SWITCH*=(*on*→*off*→*SWITCH*)"定义了一个开关类,而"*a*:*SWITCH*=(*a.on*→*a.off*→*a*:*SWITCH*)"定义了一个开关实例.进程前缀可以有多个,用于资源的标注*.如"*RESOURCE*=(*acquire*→*release*→*RESOURCE*)"定义了一类资源,而"{*a*,*b*}::*RESOURCE*=({*a.acquire*,*b.acquire*}→{*a.release*,*b.release*}→{*a*,*b*}::*RESOURCE*)"定义了由 *a* 和 *b* 共享使用的资源,其中{act_1,…,act_m}表示可以在多个动作中匹配任一动作,具体见下一节的"共享动作".

(5) 并发组合:||,表示多个简单进程间的组合.如 $P||Q$ 表示在条件允许的情况下并发执行进程 P 和 Q 中的动作序列.其中 P 和 Q 又称为子进程.

(6) 其他操作符:如变量定义、动作的优先级表示、动作的重命名以及动作隐藏等,这里不再赘述.

1.2 FSP中的并发控制

当并发执行多个进程的动作序列时,这些属于不同进程的动作间的协作关系需要某种同步设施来描述和控制.在 FSP 中有如下一些并发控制机制:

· 共享动作(shared actions).即 Barrier,指各并发子进程中共有的动作,这些动作必须在同一时刻由各相关子进程同时完成.如图 1 所示为一个产品组装的 FSP 示例.其中动作"ready"和"used"是两个共享动作,必须由子进程 MAKE_A,MAKE_B 和 ASSEMBLE 同时完成.

· 互斥(mutual exclusion).即锁机制,它的实现基于上述的共享动作,一般定义一个特殊进程:*LOCK*=(*acquire*→*release*→*LOCK*).再定义需要互斥的进程类:*WRITER*=(*acquire*→*write*→*release*→*WRITER*),组合成一个带有互斥动作的并发进程:||*CO_WRITERS*=(*a*:*WRITER*||*b*:*WRITER*||{*a*,*b*}::*LOCK*).互斥可以对应到 Java 语言的 synchronized 语句.

· 监控器和条件同步(monitors and conditional synchronization):属于锁机制的扩展,可对应于信号量操作或 Java 的 wait()…notify/notify all 语言设施,表示相应(共享)动作在给定条件(状态谓词)的监控下执行.FSP 中使用 when 操作符结合共享动作来描述条件同步.图 2 是生产者-消费者问题的 FSP 表示.

MAKE_A=(*makeA*→*ready*→*used*→*MAKE_A*),
MAKE_B=(*makeB*→*ready*→*used*→*MAKE_B*),
ASSEMBLE=(*ready*→*assemble*→*used*→*ASSEMBLE*),
||*FACTORY*=(*MAKE_A*||*MAKE_B*||*ASSEMBLE*).

Fig.1 FSP notation of the FACTORY process
图 1 FACTORY 进程的 FSP 表示

WAREHOUSE(*N*)= *GSET*[0],
GSET[*i*:0..*N*]= (when (*i*<*N*) *put*→*GSET*[*i*+1]
| when (*i*>0) *get*→*GSET*[*i*−1]),
PRODUCER=(*put*→*PRODUCER*),
CONSUMER=(*get*→*CONSUMER*),
||*PRO_CON*=(*PRODUCER* || *CONSUMER* || *WAREHOUSE*(5)).

Fig.2 FSP notation of the producer-consumer process
图 2 生产者-消费者问题的 FSP 表示

2 并发程序的两个基本属性

并发程序的正确运行包含两个方面的属性.一方面是安全性(safety),表示不能有超出规约表示的异常动作(事件)发生,或者不能进入异常状态.这里有两种情形:一种是指定事件没有发生,如信件丢失;另一种是发生了

* 这种情形在 FSP 中称为"进程共享(process sharing)".

不该发生的事件.这两种情形需要 E-CSPE 的 m 类和~类约束来描述.

另一方面是进展性(progress or liveness).并发存在多种可能性,进展性要求在条件许可的前提下可选动作集中的动作必须被执行.进展性可以再细分为若干公平(fairness)级别,如最小进展性(minimal progress)——当程序处于状态 t 时,若存在动作集合$\{act_1,act_2,\ldots,act_n\}$,其中对任一动作 act_i,t 满足 act_i 的执行条件,则这个动作集中至少有一个动作要被执行;一般公平性(weak fairness)——如果程序无限运行并多次处于上述状态 t,则上述动作集中的每一个动作都至少要被执行一次(无饿死);强公平性(strong fairness)——如果程序无限运行并进入状态 t 无限多次,则动作集中的每一个动作都必须被执行无限多次.进展性可视不同级别,采用 E-CSPE 的 a 类和 m 类约束来描述.

死锁是并发程序运行的一类特殊问题,它可以属于安全性的范畴,因为死锁代表一个异常状态,也可以属于进展性的范畴,因为这时没有动作可以被执行.可以采用 E-CSPE 的 m 类约束来描述死锁问题.

3 根据 FSP 表示推导 E-CSPE 约束

在 FSP 中,进程是并发的动作序列.每一个动作可以转化成对应的规约级事件,表示动作执行结束.事件可以定义成三元组:$e::=\langle Action,Process,Other\rangle$,其中"*Action*"为动作标识,"*Process*"为子进程标识,"*Other*"代表其他信息,如资源标识等.考虑示例"$P=(x\rightarrow L),L=(y\rightarrow STOP)$",$\langle x,P,\text{null}\rangle$事件表示进程 P 执行了 x 动作.由于进程是递归定义的,对于动作 y,对应的事件可以表示为$\langle y,P,\text{null}\rangle$或者$\langle y,L,\text{null}\rangle$.为避免二义性,事件中的进程规定为最外层定义的简单进程标识;当存在进程前缀(对应进程实例)时,进程标识为"前缀:进程名".如对 a:L,动作 $a.y$ 对应的事件将表示为$\langle y,a{:}L,\text{null}\rangle$.

3.1 简单进程中E-CSPE约束的推导

下面基于 FSP 的基本操作符(不包括||)考虑 E-CSPE 约束的推导规则.

(1) 顺序:$P=x\rightarrow Q$.对$\forall e\in FirstAction(Q)$,其中 $FirstAction(Q)$表示 Q 中第 1 个可能动作所对应的事件集;令 C_e 表示控制事件(或动作)e 是否发生的状态谓词(如上述的 when 条件式),这里 C_x=true.有 $a[[\langle x,P,\text{null}\rangle;\rightarrow e]](C_e)$.

若 P 是对应整个程序的进程定义,则有

$$a[[\#;\rightarrow\langle x,P,\text{null}\rangle]](\text{true}),\ \text{令\#事件标识程序的启动.}$$

(2) 选择:$P=x\rightarrow Q|y\rightarrow R$.则

对$\forall e\in FirstAction(Q)$,有 $a[[\langle x,P,\text{null}\rangle;\rightarrow e]](C_e)$;

且对$\forall e\in FirstAction(R)$,有 $a[[\langle y,P,\text{null}\rangle;\rightarrow e]](C_e)$.

若 P 是针对整个程序的进程定义,则有

$$a[[\#;\rightarrow\langle x,P,\text{null}>]](\text{true});\text{及 }a[[\#;\rightarrow\langle y,P,\text{null}\rangle]](\text{true}).$$

(3) 条件控制:P=when B $x\rightarrow Q|y\rightarrow R$.此时 $C_x=B,C_y$=true.则

对$\forall e\in FirstAction(Q)$,有 $a[[\langle x,P,\text{null}\rangle;\rightarrow e]](C_e)$;

对$\forall e\in FirstAction(R)$,有 $a[[\langle y,P,\text{null}\rangle;\rightarrow e]](C_e)$.

若 P 是针对整个程序的进程定义,则有

$$a[[\#;\rightarrow\langle x,P,\text{null}\rangle]](B);\text{及 }a[[\#;\rightarrow\langle y,P,\text{null}\rangle]](\text{true}).$$

(4) 进程前缀:考虑开关程序 a:$SWITCH=(a.on\rightarrow a.off\rightarrow a{:}SWITCH)$,可推出:

$$a[[\#;\rightarrow\langle on,a{:}SWITCH,\text{null}\rangle]](\text{true}),$$
$$a[[\langle on,a{:}SWITCH,\text{null}\rangle;\rightarrow\langle off,a{:}SWITCH,\text{null}\rangle]](\text{true}),$$
$$a[[\langle off,a{:}SWITCH,\text{null}\rangle;\rightarrow\langle on,a{:}SWITCH,\text{null}\rangle]](\text{true}).$$

3.2 并发进程中E-CSPE约束的推导

下面基于 FSP 的并发控制机制考虑 E-CSPE 约束的推导规则:

(1) 并发组合:$||P\_Q=P||Q$.对$\forall e\in FirstAction(P)\cup FirstAction(Q)$,有

$$a[[\#;\rightarrow e]](C_e).$$

(2) 共享动作.共享动作主要考虑的是并发程序的安全属性,应描述为 m 类和~类约束.具体是:考虑如图 1 所示的 *FACTORY* 进程.

针对动作"*ready*",设 e 代表任一事件,有:

对单个进程 *MAKE_A*(*MAKE_B* 或 *ASSEMBLE*),

~[[⟨*ready*,*MAKE_A*,null⟩;→e]](at least one of the processes,*MAKE_B* or *ASSEMBLE*, does not do the *ready*);

对并发进程||*FACTORY*,设 i,j 分别代表进程 *MAKE_A* 或 *MAKE_B* 或 *ASSEMBLE*,有

m[[⟨*ready*,i,null⟩;→⟨*ready*,j,null⟩]]($i\neq j\wedge$process j does not do the *ready*).

针对动作"*used*"可以做类似推导.

(3) 互斥.即定义锁进程 *LOCK*,如上节的"*CO_WRITERS*"进程.互斥同时涉及并发程序的安全性和进展性.针对安全属性有

~[[⟨*acquire*,i,null⟩;→⟨*acquire*,j,null⟩]](the lock is not released, i.e. ⟨*release*,i,null⟩ does not happen),

~[[⟨*release*,i,null⟩;→⟨*release*,j,null⟩]](the lock is not assigned, i.e. ⟨*acquire*,j,null⟩ does not happen),

其中 i 和 j 分别代表进程 *a*:*LOCK* 或 *b*:*LOCK*."*a*:*LOCK*"的含义可以理解为进程实例 *a* 执行进程 *LOCK* 的动作,或者说在进程 *LOCK* 的"环境"下执行动作.针对进展属性有

a[[⟨*acquire*,i,null⟩;→⟨*release*,i,null⟩]](true).

进展属性有不同的公平级别.考虑最小进展性,有

a[[#;→⟨*acquire*,i,null⟩]](true),

a[[⟨*release*,i,null⟩;→⟨*acquire*,i,null⟩]](true);

考虑一般公平性(no starvation),需增加

m[[#;→⟨*acquire*,i,null⟩]](true);

考虑强公平性,还需增加

m[[⟨*release*,i,null⟩;→⟨*acquire*,i,null⟩]](true).

互斥机制存在死锁问题,死锁应使用 m 类约束,即

m[[⟨*acquire*,i,null⟩;→⟨*release*,i,null⟩]](true).

(4) 条件同步:在 FSP 中条件同步是锁机制的扩展,其实现基于共享动作,描述上使用 when 操作符和状态变量定义.与互斥机制相同,条件同步同时涉及并发程序的安全和进展属性.考虑如图 2 所示的生产者-消费者进程.就"*put*"动作(事件),针对安全属性有

~[[#;→⟨*put*,*p*:*WAREHOUSE*,null⟩]](¬(i<N)),

~[[⟨*put*,*p*:*WAREHOUSE*,null⟩;→⟨*put*,*p*:*WAREHOUSE*,null⟩]](¬(i<N)),

~[[⟨*get*,*c*:*WAREHOUSE*,null⟩;→⟨*put*,*p*:*WAREHOUSE*,null⟩]](¬(i<N));

针对进展属性,考虑一般公平性有

a[[#;→⟨*put*,*p*:*WAREHOUSE*,null⟩]](i<N),

a[[⟨*put*,*p*:*WAREHOUSE*,null⟩;→⟨*put*,*p*:*WAREHOUSE*,null⟩]](i<N),

a[[⟨*get*,*c*:*WAREHOUSE*,null⟩;→⟨*put*,*p*:*WAREHOUSE*,null⟩]](i<N).

就"*get*"动作(事件)可以作类似推导.

按照上述推导规则,可以根据并发程序的 FSP 表示定义出用于测试的 E-CSPE 约束.当 FSP 表示本身难以完备时,可以就一般性的规约要求导出 E-CSPE 约束.例如对安全属性,可就任一异常事件 *error*,定义~[[#;→*error*]](true);对进展属性,视不同的公平性级别,可就一个必然事件 *milestone* 定义 a[[#;→*milestone*]](true)或者 m[[#;→*milestone*]](true)等.利用已有的约束,可按照 E-CSPE 约束之间的关系[2]再推导出新的有效约束.

4 实例研究

考虑生产者-消费者问题,修改||*PRO_CON* 进程的定义如下:

||*PRO_CON*=(*pro*[1..3]:*PRODUCER*||*con*[1..3]:*CONSUMER* || {*pro*[1..3],*con*[1..3]}::*WAREHOUSE*(5)).

现在 *PRODUCER* 对应 3 个生产者实例,其动作为“*put*”;*CONSUMER* 对应 3 个消费者实例,动作为“*get*”.*WAREHOUSE* 的最大容量为 *N*(*N*=5),初始状态为空“null”.3 类进程并发组合在一起构成生产者-消费者程序.程序中涉及并发组合、共享动作以及条件同步.

FSP 的一个主要特点是可以较容易地转换成对应的 Java 程序,如上述的 WAREHOUSE 可以实现为一个共享对象 *WareHouse*,动作“*put*”和“*get*”实现为对象方法.PRODUCER 和 CONSUMER 可以实现为线程对象,主要的动作分别是“*put*”和“*get*”.如图 3 所示为仓库的实现(基于 RMI 机制),“**synchronized**”保证该段代码在给定时刻最多由一个线程来执行.

```
public interface WareHouse extends Remote {…}
class WareHouseImpl extends UnicastRemoteObject
    implements WareHouse {
    Object gSet[5]; …
    public synchronized void put(Object o)
        throws InterruptedException {
        while (count==5) wait();
        gSet[in]=o; count++; in=(in+1)% 5;
        notify();
    }
    public synchronized Object get()
        throws InterruptedException {
        while (count==0) wait();
        Ojbect o=gSet[out];
        gSet[out]=null; count--; out=(out+1)% 5;
        notify();
        return o;
    } }
```

Fig.3 Java Implementation for the WAREHOUSE

图 3 WAREHOUSE 的 Java 实现

根据上一节的推导规则可以推导出以下 32 个用于测试的 E-CSPE 约束,其中 $1\le i,j\le 3$,?代表任意一个生产者或消费者.

(1) *a*[[#;→⟨*put*,*pro*[*i*]:*P*,null⟩]](true).

(2) *a*[[⟨*put*,*pro*[*i*]:*P*,null⟩;→⟨*put*,*pro*[*i*]:*P*,null⟩]](true).

(3) *a*[[#;→⟨*get*,*con*[*j*]:*C*,null⟩]](true).

(4) *a*[[⟨*get*,*con*[*j*]:*C*,null⟩;→⟨*get*,*con*[*j*]:*C*,null⟩]](true).

(5) *a*[[#;→⟨*put*,*pro*[?]:*W*,null⟩]](*content*<*N*).

(6) *a*[[#;→⟨*get*,*con*[?]:*W*,null⟩]](*content*>0).

(7) *a*[[⟨*put*,*pro*[?]:*W*,null⟩;→⟨*put*,*pro*[?]:*W*,null⟩]](*content*<*N*).

(8) *a*[[⟨*put*,*pro*[?]:*W*,null⟩;→⟨*get*,*con*[?]:*W*,null⟩]](*content*>0).

(9) *a*[[⟨*get*,*con*[?]:*W*,null⟩;→⟨*get*,*con*[?]:*W*,null⟩]](*content*>0).

(10) *a*[[⟨*get*,*con*[?]:*W*,null⟩;→⟨*put*,*pro*[?]:*W*,null⟩]](*content*<*N*).

(11) *m*[[⟨*put*,*pro*[*i*]:*P*,null⟩;→⟨*put*,*pro*[*i*]:*W*,null⟩]]((⟨*put*,*pro*[*i*]:*W*,null⟩ did not happen)∧(*content*<*N*)).

(12) *m*[[⟨*get*,*con*[*j*]:*C*,null⟩;→⟨*get*,*con*[*j*]:*W*,null⟩]]((⟨*get*,*con*[*j*]:*W*,null⟩ did not happen)∧(*content*>0)).

(13) ~[[⟨*put*,*pro*[*i*]:*P*,null⟩;→⟨*put*,*pro*[*i*]:*P*,null⟩]](⟨*put*,*pro*[*i*]:*W*,null⟩ did not happen within).

(14) ~[[⟨*get*,*con*[*j*]:*C*,null⟩;→⟨*get*,*con*[*j*]:*C*,null⟩]](⟨*get*,*con*[*j*]:*W*,null⟩ did not happen within).

(15) *m*[[⟨*put*,*pro*[*i*]:*W*,null⟩;→⟨*put*,*pro*[*i*]:*P*,null⟩]](⟨*put*,*pro*[*i*]:*P*,null⟩ did not happen).

(16) *m*[[⟨*get*,*con*[*j*]:*W*,null⟩;→⟨*get*,*con*[*j*]:*C*,null⟩]](⟨*get*,*con*[*j*]:*C*,null⟩ did not happen).

(17) ~[[⟨*put*,*pro*[*i*]:*W*,null⟩;→⟨*put*,*pro*[?]:*W*,null⟩]](⟨*put*,*pro*[*i*]:*P*,null⟩ did not happen within).

(18) ~[[⟨*get*,*con*[*j*]:*W*,null⟩;→⟨*get*,*con*[?]:*W*,null⟩]](⟨*get*,*con*[*j*]:*C*,null⟩ did not happen within).

(19) ~[[⟨*put*,*pro*[*i*]:*W*,null⟩;→⟨*get*,*con*[?]:*W*,null⟩]](⟨*put*,*pro*[*i*]:*P*,null⟩ did not happen within).

(20) ~[[⟨*get*,*con*[*j*]:*W*,null⟩;→⟨*put*,*pro*[?]:*W*,null⟩]](⟨*get*,*con*[*j*]:*C*,null⟩ did not happen within).

(21) ~[[#;→⟨*put*,*pro*[?]:*W*,null⟩]](*content*≥*N*).

(22) ~[[#;→⟨*get*,*con*[?]:*W*,null⟩]](*content*≤0).

(23) ~[[⟨*put*,*pro*[?]:*W*,null⟩;→⟨*put*,*pro*[?]:*W*,null⟩]](*content*≥*N*).

(24) ~[[⟨*get*,*con*[?]:*W*,null⟩;→⟨*get*,*con*[?]:*W*,null⟩]](*content*≤0).

(25) ~[[⟨*get*,*con*[?]:*W*,null⟩;→⟨*put*,*pro*[?]:*W*,null⟩]](*content*≥*N*).

(26) ~[[⟨*put*,*pro*[?]:*W*,null⟩;→⟨*get*,*pro*[?]:*W*,null⟩]](*content*≤0).

(27) *a*[[#;→⟨*put*,*pro*[?]:*W*,null⟩]](*content*<*N*).

(28) *a*[[#;→⟨*get*,*con*[?]:*W*,null⟩]](*content*>0).

(29) *a*[[⟨*put*,*pro*[?]:*W*,null⟩;→⟨*put*,*pro*[?]:*W*,null⟩]](*content*<*N*).

(30) *a*[[⟨*get*,*con*[?]:*W*,null⟩;→⟨*get*,*con*[?]:*W*,null⟩]](*content*>0).

(31) *a*[[⟨*get*,*con*[?]:*W*,null⟩;→⟨*put*,*pro*[?]:*W*,null⟩]](*content*<*N*).

(32) *a*[[⟨*put*,*pro*[?]:*W*,null⟩;→⟨*get*,*pro*[?]:*W*,null⟩]](*content*>0).

观察FSP的语义不难发现,对于Java程序,共享动作对应于对象间的方法调用.如事件⟨*put*,*pro*[*i*]:*P*,null⟩相当于线程 *pro*[*i*]调用"*wareHouse*.*put*(…)",而事件⟨*put*,*pro*[*i*]:*W*,null⟩相当于对象方法"*WareHouseImpl*::*put*(…)"的执行.另外,考虑程序状态时可以引入状态常量(invariant),如"*content*≤*N*"及"*content*≥0".根据上述语义可以调整E-CSPE约束的定义.

考虑事件序列 *s* 和事件约束 *r*.*s* 可能覆盖 *r*,可能违反 *r*,亦可能同 *r* 无关.据此可定义E-CSPE约束间的一致性.推导出的E-CSPE约束彼此之间可能互相冲突,也可能互相重复,需要根据规约调整不一致的情形.

5 结束语和相关工作比较

Jeff Magee等人[3]利用FSP记法基于有限状态机来描述并发进程,并根据进程代数的演算验证程序表示的正确性.采用这类形式化验证存在两方面的问题:一方面是代数演算限制了程序的规模;另一方面是即使证明FSP表示完全无误,也不能保证对应的程序实现完全正确.

Carver等人[4]基于程序实现,按照源程序的语法树(semantic graph)推导事件约束.推导出的CSPE约束实际上描述了程序运行时刻所产生的可行事件序列的基本特性,可以作为判断测试充分性的基准.但它缺乏从规约角度出发的验证.

本文从规约角度出发,基于并发程序的FSP表示推导用于测试的E-CSPE约束.FSP表示可以较容易地转换成Java程序.鉴于目前Internet已成为最大的并得到广泛接受的分布计算环境,而Java又是Internet上的标志性语言,选择Java程序作为测试对象有一定的意义.

在本文工作的基础上,需要进一步研究的问题包括:如何将规约级事件映射到语言级甚至实现级事件[5],并根据程序实现精化事件约束的定义;如何增强 E-CSPE 约束推导和应用的实用性,如加大被测程序规模,推出原型系统等;如何考虑其他类型的编程语言(如C++)和分布式体系结构如(CORBA),并进行对应的E-CSPE约束推导等问题.

References:

[1] Carver, R.H., Tai, Kou-Chung. Use of sequencing constraints for specification-based testing of concurrent programs. IEEE Transactions on Software Engineering, 1998,24(6):471~490.

[2] Gu, Qing, Chen, Dao-xu, Yu, Meng, Xie, Li, *et al*. Validation test of distributed program based on events sequencing constraints. Journal of Software, 2000,11(8):1035~1040 (in Chinese).

[3] Magee, J., Kramer, J. Associated Concurrency: State Models & Java Programs. Wiley, 1999.

[4] Carver, R.H., Tai, Kou-Chung. Static analysis of concurrent software for deriving synchronization constraints. In: Proceedings of the 11th International Conference on Distributed Computing Systems (ICDCS'91). IEEE Computer Society Press, 1991. 544~551. http://www.mpi-sb.mpg.de/services/library/proceedings/contents/icdcs91.html.

[5] Gu, Qing, Chen, Dao-xu, Han, Jie, *et al*. A system framework for distributed programming test. Journal of Software, 2000,11(8): 1053~1059 (in Chinese).

附中文参考文献:

[2] 顾庆,陈道蓄,于勐,等.基于事件约束的分布式程序正确性测试.软件学报,2000,11(8):1035~1040.

[5] 顾庆,陈道蓄,韩杰,等.一个面向分布式程序的测试系统框架.软件学报,2000,11(8):1053~1059.

Event Constraints Definition Based on Finite State Process*

GU Qing, CHEN Dao-xu, XIE Li, HAN Jie, SUN Zhong-xiu

(*State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing* 210093, *China*)

E-mail: guq@dislab.nju.edu.cn

http://www.nju.edu.cn

Abstract: When a distributed program is under test, event sequencing constraints should be defined to check the event sequences generated after the program was executed. Those event constraints shall be derived from specifications of the program. FSP is a kind of process algebra notation that can be used to describe formal specifications of concurrent programs. FSP describes concurrent processes as action sequences, where an action can be mapped to a specification-level event. The E-CSPE constraints define the sequential relationship between any two runtime events under given state predicates. Based on the operators and concurrency control facilities within FSP, the E-CSPE constraints can be derived. Those derived E-CSPE constraints consider the safety and liveness properties of the concurrent program, and based on them both the correctness of the program execution and the sufficiency of the test work can be judged.

Key words: software testing; finite state process; specification-based testing; concurrent programs; event sequencing constraints

* Received March 6, 2001; accepted August 1, 2001

Supported by the Key Science-Technology Project of the National 'Ninth Five-Year-Plan' of China under Grant No.98-780-01-07-03