第6卷第3期
1995年3月

软 件 学 报
JOURNAL OF SOFTWARE

Vol. 6, No. 3
Mar. 1995

模拟式实时语义*

韩 燕

(中国科学院软件研究所,北京 100080)

摘要 程序的实时行为依赖于其所在的实时环境,诸如所采用的编译程序、调度程序、通信媒介以及主机等.本文设法用模拟环境的手段来刻化程序的实时行为,从而为实时语义提供了一种通用的方法.文中所采用的逻辑是通常的线性时态逻辑,但引入了各种时间变量.

关键词 实时语义,实时通信顺序进程,时态逻辑,模拟,调度.

程序的实时行为随其所在的环境而变化,若使用一个优化的编译程序,则可减少执行程序语句所需的机器的周期数;使用优先调度程序,高优先度的程序可抢占处理机,优先运用;同机内通信或远程通信所花费的时间又有巨大区别;自然,主机的速率也是程序实时行为的一个决定性因素.近年来各种语言的非实时语义日趋标准化,各个编译程序都必须遵循标准语义.而实时语义的标准化的说法似乎未曾听说,其原因之一,恐怕就是实时语义对环境的依赖性.

程序的实时行为又是程序应用中至为关键的问题,特别在严格时间系统(time critical system)中,差之毫秒,失之千里.时间上的差错会使两列火车碰撞、两艘飞船无法对接,以至各种不可想象的灾难事故.程序的实时行为的研究已提到日程.文献[1—6]都从各种不同角度讨论实时行为的描述.文献[1]提到了程序行为对环境的依赖性,但仅对调度程序对实时行为的影响作了深一步的探讨.其它论文则对环境作了各种假设,设想在极大并行(maxium parallelism)条件下程序的实时行为.其结果只考虑了程序间通信同步对实时行为的影响,而暂时略去其它实时因素.

本文将用模拟的方法讨论各种环境因素对程序实时行为的影响.由于调度程序的实时行为已在文献[1]中讨论了,故本文将略去不提.

就编译程序而言,我们将用函数 *evaluation*(简写为 *ev*)模拟表达式赋值所需的计算机运行周期数,故函数 *ev* 是一个由表达式和当前变元状态至自然数的函数,即

$$ev : Exp \times State \rightarrow Nat.$$

操作系统对通道的管理也消耗时间,函数 *monitor*(简写为 *mon*)模拟通道管理的计算机周期数,故 *mon* 是一个由通道端至自然数的函数,即

$$mon : Channel - Ends \rightarrow Nat.$$

* 本文1993—03—01收到,1993—05—25定稿

作者韩燕,1969年生,1993年硕士毕业于中国科学院软件研究所,主要研究领域为实时形式系统的语义描述.

本文通讯联系人:周巢尘,北京 100080,中国科学院软件研究所

计算机的速率记作 μ. μ 是一个正实数,其单位也许是毫微秒或者微微秒. 在状态 v 下,对表达式 e 赋值所消耗的时间将为 $\mu * ev(e,v)$. 操作系统管理通道 c 的输入(记作 $c?$)所花时间,则为 $\mu * mon(c?)$;管理 c 的输出(记作 $c!$)所需时间,则为 $\mu * mon(c!)$.

上述的模拟不尽完善,例如未曾考虑传输数据量对通道管理的影响等. 本文的目的是解释这一模拟方法,不宜涉及过多细节.

文献[2]中建议使用带有时间变量的线性时态逻辑定义语言的实时行为. 一般的时态逻辑只能确定程序各个操作的时间上的先后次序,不能给出实际的操作时间,将时间变量作为一种新的时态变量,就可记录操作发生的实际时刻,描述程序的实时行为. 度量时态逻辑[3]和区段逻辑[1]通过说明两个操作间的时间差来定义程序的实时行为. 前者使用时灵便,后者由于抽象而使公式简化. 本文中采用带有时间变量的时态逻辑. 至于各种逻辑的优劣则不属本文讨论的范围.

线性时态逻辑的数学模型是一组变量,它们在不同的观测时刻可取不同的值. 这组变量称为时态变量,而观测时刻具有线性序,可离散、可连续,可及过去和未来. 本文中只用离散序,只及未来. 故本文中时态变量可看作是自然数上的一个函数,但函数的自变元(观测时刻)略去不写,而用时态算子 O(下一观测时刻)及$\Diamond$(未来某一观测时刻)和$\Box$(未来任意观测时刻)来表示所涉及的观测时刻. 例如,时态变量 x 表示某一程序变元在各个观测时刻的值. 则 $x=1$ 说明当前观测时刻该变元取值 1;$Ox=2$ 说明下一个观测时刻该变元的取值为 2;而$\Diamond(x=100)$表示未来某个观测时刻 x 的值为 100;至于$\Box(x=1)$则说明该变元从本观测时刻起,永不改变,总取值为 1.

除去一元时态算子外,还有二元算子 **until** 和 **unless**,(A **until** B)的含义是,在未来的某个观测时刻公式 B 成立,而在该时刻前,A 一直成立. (A **unless** B) 的含义更广泛,包含(A **until** B),而且还允许 B 永不成立,而 A 永远成立.

时态逻辑及其在程序理论中的应用可参阅文献[7,8],这里不作介绍.

1　程序语言

我们用一个简单的 CSP(Communicating Sequential Processes)语言来解释我们的模拟式实时语义. 这个语言中包含有赋值语句、通信语句、并发语句、延迟语句以及循环语句等必要成分.

语法:

(1)进程项(*term*)

$$P::=p\mid STOP\mid c?x\rightarrow P\mid c!e\rightarrow P\mid wait\ e\rightarrow P\mid$$
$$(c?x\rightarrow P\,\Box\,d?y\rightarrow Q)\mid(c?x\rightarrow P\,\Box\,wait\ e\rightarrow Q)$$

P 代表任意进程名,$STOP$ 表示终止,$c?\ x$ 表示从通道 c 输入信息、并存入变元 x 处,$c!\ e$ 表示由通道 c 发出表达式 e 的值,$wait\ e$ 表示等待 e 时间,$\Box$ 为通常的选择算子,$\rightarrow$则可理解为顺序算子.

(2)顺序进程

$$S::=(p\triangleq P)$$

进程项 P 中可出现进程名 p,此时 $p\triangleq P$ 为递归定义. 例如 $p\triangleq(c!\ 1\rightarrow p)$,定义了一个

不断在 c 通道上发出信息 1 的进程. 假设通道都是单向的. S 是顺序进程,而通信只能在并发进程间交换信息,故对任意通道 c,S 中不能同时出现 c 的输入和输出命令.

(3)并发进程

$$C::=S\|S|S\|C$$

并发进程是由顺序进程组成的并行系统. 这里假设同一系统的顺序进程间不共享变元,也不共享通道端,即不考虑广播式通信. 例如

$S1:p1\triangle c!\ 1\rightarrow p1$

$S2:p2\triangle c?\ x\rightarrow p2$

$S3:p3\triangle c?\ y\rightarrow STOP$

而$(S1\|S2)\|S3$是非法的,因为 $S2$,$S3$ 间共享 c 的输入端.

使用这一语言,我们可以书写不少有趣的实时程序. 例如,发送进程 *sender* 可表示为

$$p1\triangle user?x\rightarrow medium!x\rightarrow(ack?y\rightarrow p1\,\square wait\ 1\rightarrow STOP).$$

sender 经通道 *user* 向用户收集数据,然后经介质 *medium* 发出数据. 数据一旦传出,*sender* 就计时等待确认信息. 若确认信息在预定时间 1 分钟内到达,就向用户收集下一个数据;否则,作为传输线断连而终止.

当然,这不是一个实用的实时语言,我们只是以它为例解释模拟式实时语义.

2 时 钟

通道端(输入端/输出端)记录着通信的起止,不妨假设设置有时钟. 以时态变量 $t_{c?}$ 和 $t_{c!}$ 表示 c 的输入端和输出端时钟值. 每个顺序进程 S,自然亦有一时钟记录各个操作的执行和终止时间. 令时态变量 t_s 表示进程 S 的时钟值. 若 S 拥有一个通道端,有理由假设 S 要求通道端的时钟和 S 的进程时钟同步. 令 $\alpha(S)$ 为进程 S 的通道端集合. 对时钟间的这一同步假设可用时态公式表示:

$$\square(t_{c?}=t_s)\quad(c?\in\alpha(S)),\qquad\square(t_{c!}=t_s)\quad(c!\in\alpha(S)).$$

令 SYN_s 表示 S 中所有通道端时钟(仅有有穷个)和 S 的进程时钟的同步假设的合取式,即

$$SYN_s=_{df}\bigwedge_{c\#\in\alpha(s)}\square(t_{c\#}=t_s)\qquad(\#\in\{?,!\})$$

这样,公式 SYN_s 定义了 S 中各个时钟的同步关系.

并发进程间实行通信,而 *CSP* 式的语言假定同步通信为其唯一通信机制. 故可假设位于同一并发进程系统中的各个顺序进程的时钟在启动时必须同步. 设 C 为 $S1\|S2\|\cdots\|Sn$,则令

$$SYN_c=_{df}\bigwedge_{i=1}^{n-1}(t_{si}=t_{si+1})$$

SYN_c 定义了 C 中各个进程时钟的启动同步.

这种同步假设并不限制进程间启动的可能先后. 设 $S1$ 和 $S2$ 共处于一个并发系统,但 $S2$ 在 $S1$ 启动 1 小时后才启动. 这可用$(wait\ 60\rightarrow S2)$取代 $S2$,并设想取代后的两个进程时钟在启动时即同步.

3 实时语义

给定顺序进程 S,令 V_s 为 S 的状态空间,即 S 中变量 X_s 至值域 V 的一个函数.例如,进程 S 为 $p\triangle(c?x\rightarrow d!(y+1)\rightarrow STOP)\ \square\ (wait(x+2)\rightarrow p)$,则 $\alpha(S)=\{c?,d!\}$,$X_s=\{x,y\}$,$v_s: X_s\rightarrow V$.

为定义 S 的实时语义,对 $\alpha(S)$中的每个元素 $c?$ 或 $d!$,引入相应的时态变量,在不致引起混淆的情况下,即记之为 $c?$ 及 $d!$.$c?$ 及 $d!$ 在不同的观测时刻可取不同的值.其取值范围为 $V\cup\{tt,ff\}$,其中 tt,ff 不属于 V.$c?$ 取值为 tt,表示通道 c 的输入端已就绪,准备随时接受自通道 c 的输出端发来的数据;$c?$ 取值为 ff,表示 c 的输入端此时无法接收数据.类似地,$c!$ 取值为 tt,表示 c 的输出端已就绪,随时可输出数据;$c!$ 取值为 ff,则表示输出端此时不准备输出数据.由于是采用同步通信,c 通道上的通信只有在其输入和输出端同时就绪时才能发生.也就是,只有当 $c?$ 和 $c!$ 同时取值为 tt 时,通信才能发生.当通信发生时,我们令 $c!$ 和 $c?$ 的取值为通信时传递的数据,即 $c!,c?\in V$,而且相等.这样,我们使用时态变量 $c?$ 和 $c!$ 完全刻划了通道 c 上的通信过程.而通信的实际时刻,则用 $c?$ 和 $c!$ 的时钟 $t_{c?}$ 及 $t_{c!}$ 所记录的各自准备就绪时刻的极大值来表示.这个极大值就是双方同时就绪的时刻,也就是通信实际可发生的时刻.由于通信介质和传递的数据量都会影响进程的实时行为,故引入函数 $pass$

$$pass: Channels\times V\rightarrow Reals,$$

$pass(c,m)$ 是一个正实数,模拟在通道 c 上传递数据 m 所消耗的实际时间.

上述说明中,通信的同步机制可形式化为

$$SYN=_{df}\ \square\ \bigwedge_{c\in channels}((c?=m)\Leftrightarrow(c!=m)),\qquad m\in V.$$

下面我们将转入定义第 1 节中所给定的语言的实时语义.

(1)$STOP$.设 $STOP$ 的进程名为 S.$STOP$ 的含义为终止,故

$$【STOP】=_{df}\ \square(\alpha(S)=ff\wedge Ot_s=t_s\wedge Ov_s=v_s).$$

也就是 S 的各个通道端不再要求通信(我们用 $\alpha(S)=ff$ 表示 $\bigwedge_{c\#\in\alpha(S)}(c\#=ff)$),时钟不再记录(即 $Ot_s=t_s$),程序变元值不再改变(即 $Ov_s=v_s$).

(2)$c?\ x\rightarrow P$.设该进程名为 S.$(c?\ x\rightarrow P)$的含义为 S 先在 c 通道上接收数据,然后动作如 p,故其语义可分为 4 部分.

第 1 部分是等待操作系统处理通道 c 请求输入,可形式化为

$$\alpha(S)=ff\wedge Ot_s=t_s\wedge Ov_s=v_s\wedge O\alpha(S)=\alpha(S).$$

此处$(O\alpha(S)=\alpha(S))$亦是一个简写,表示 $\bigwedge_{c\#\in\alpha(S)}Oc\#=c\#$ 即 $c\#$ 的状态不改变.上述公式记作 $Idle$.

第 2 部分为 c 通道输入就绪,可表达为

$$O\alpha(S)=\alpha(S)[tt/c?]\wedge Ots=ts+\mu*mon(c)?\wedge Ov_s=v_s.$$

此处 $\alpha(S)[tt/c?]$表示将 $c?$ 的值代之为 tt,$\alpha(S)$中其它变量的值不变;也就是 c 的输入端就绪而 S 中其它通道无输入输出要求.而 t_s 记载了 c 的输入端就绪的时刻,也就是启动时刻加上操作系统所消耗的处理通道请求时间,即 $\mu*mon(c?)$;而程序变量值自然没有改变,即 $Ov_s=v_s$,这一公式记作 $Ready(c?)$.

第3部分则表示 S 等待输出方响应,以实现通信.可由公式

$$O\alpha(S)=\alpha(S)\wedge Ots=ts\wedge Ov_s=v_s.$$

表示.即 S 不改变状态,一味等待,记作 *Wait*.

第4部分表示通信同步,S 接收输入并转向执行 P.可以表示为

$$\exists m.c!=tt\wedge Ot_s=(\max(t_s,t_{c!})+pass(c,m))\wedge O\alpha(S)=\alpha(S)[m/c?]$$
$$\wedge Ov_s=v_s[m/x]\wedge O【P】.$$

也就是 c 的输出端也已就绪,即 $c!=tt$.接着数据 m 由通道 c 传至 S,t_s 记载了由同步起至通信完成的时刻,即 $\max(t_s,t_{c!})+pass(c,m)$.$S$ 接收数据 m 后,将 m 存入程序变量 x 所对应的单元,然后转向 P.改公式记作 $Pass(c?,P)$.

进程($c?\ x\to P$)的语义为这四部分的合成,即

$$【c?x\to P】=_{df}\ Idle\ \textbf{until}(Ready(c?)\wedge O(Wait\ unless\ Pass(c?,P)))$$

Wait 和 *Pass* 间用 *unless* 连接,因为这种等待可能永不成功,称之为死等或死锁.死锁的原因之一,为 c 的输出方永不执行输出命令,从而导致输入方死锁.

(3)$c!\ e\to P$.设该进程名为 S.($c!\ e\to P$)的语义和($c?\ x\to P$)十分相似,只是第2部分输出就绪 $Ready(c!,e)$,应为

$$O\alpha(S)=\alpha(S)[tt/c!]\wedge Ot_s=t_s+\mu*mon(c!)+\mu*ev(e,vs)\wedge Ov_s=v_s.$$

此处待输出的表达式 e 的赋值时间亦已计入.

第4部分 $Pass(c!,P)$ 为

$$c?=tt\wedge Ot_s=(\max(t_s,t_{c?})+pass(c,e(v_s)))\wedge Ov_s=v_s$$
$$\wedge O\alpha(S)=\alpha(S)[e(v_s)/c!]\wedge O【P】.$$

此处 $e(v_s)$ 表示表达式 e 在状态 v_s 下的实际值.($c!\ e\to P$)的语义为

$$【c!\ e\to P】=_{df}\ Idle\ \textbf{until}(Ready(c!,e)\wedge O(Wait\ unless\ Pass\ (c!,p)))$$

(4)$wait\ e\to P$.设该进程名为 S.

$$【wait\ e\to P】=_{df}\ Idle\ \textbf{until}(O\alpha(S)=\alpha(S)\wedge Ot_s=t_s+\mu*ev(e,v_s)+e(v_s))$$
$$\wedge Ov_s=v_s\wedge O【P】$$

其中实际等待时间是要求等待时间 $e(v_s)$ 外加 e 的赋值时间 $\mu*ev(e,v_s)$.

(5)($c?\ x\to P\ \square\ d?\ y\to Q$).设其进程名为 S.语句的含义为 S 在 c,d 上等待输入,选择首先响应者,转入相应的分支.S 的语义即分4部分.*Idle* 和 *Wait* 部分和前面的相同.*Ready* 部分则为在两个通道上同时就绪,即令 $Ready(c?,d?)$ 为

$$O\alpha(S)=\alpha(S)[tt/c?][tt/d?]\wedge Ot_s=t_s+\mu*mon(c?)+\mu*mon(d?)\wedge Ov_s=v_s$$

而且 *Pass* 部分更为复杂.若选择第一分支则要求 c 通道的同步时间不迟于 d 通道的同步时间,即为

$$Pass(c?,P)\wedge(d!=tt\Rightarrow t_{c!}\leqslant t_{d!})$$

这个附加条件说明,d 的输出端的响应时间不先于 c 的输出端(即 $t_{c!}\leqslant t_{d!}$).类似地需要增加选择第2分支的条件

$$Pass(d!,Q)\wedge(c!=tt\Rightarrow t_{d!}\leqslant t_{c!})$$

这两个公式的析取式记作 $Pass(c?,P,d?,Q)$.则

$$【c?x\to P\ \square\ d?y\to Q】=_{df}\ Idle\ \textbf{until}(Ready(c?,d?))$$

$$\wedge\ O(Wait\ unless\ Pass(c?,P,d?,Q))$$

(6)$(c?\ x\rightarrow P\ \square wait\ e\rightarrow Q)$. 设其进程名为 S. 语句的含义是 S 在 c 上等待输入,若输出方的响应迟于设置的时限 e,则 s 选择第 2 分支.

故选择第 1 分支时,S 的行为满足

$$Pass(c?,P)\ \wedge\ t_{c!}\leqslant(t_s+\mu*ev(e,v_s)+e(v_s)),$$

而选择第 2 分支时,S 的行为应满足

$$c!=ff\ \wedge\ O\alpha(S)[ff/c?]\ \wedge\ Ot_s=(t_s+\mu*ev(e,v_s)+e(v_s))\ \wedge\ Ov_s=v_s\ \wedge\ O[\![P]\!].$$

上述两公式的析取式记作 $Pass(c?,P,e,Q)$. 则

$$[\![c?x\rightarrow P\ \square wait\ e\rightarrow Q]\!]=_{df}Idle\ \textbf{until}\ (Ready(c?)\ \wedge\ O(Wait\ \textbf{until}\ Pass(c?,P,e,Q))).$$

和$(c?\ x\rightarrow P\ \square d?\ y\rightarrow Q)$的语义相比,最后一个时态连接词已改为 **until**. 因为$(c?\ x\rightarrow P\ \square wait\ e\rightarrow Q)$总会成功地转向执行 P 或 Q 之一.

(7)$p\triangleq P$. 递归式的语义,可由传统的不动点方法给出. 在时态逻辑中定义偏序

$$(A\supseteq B)=_{df}\ \ (A\Rightarrow B),$$

故

$$\perp=_{df}\ \ true.$$

递归式的语义就可定义为

$$\forall\ n\ P^n(true).$$

此处

$$P^0(true)=_{df}true$$

$$P^{n+1}(true)=_{df}P[P^n(true/p)].$$

(8)并发进程. 以$(S1\parallel S2)$为例.

在第 2 节中我们已提到并发系统中的启动时钟同步的假设,也就是 $SYN_c=_{df}(t_{s1}=t_{s2})$. 此外,我们还希望避免两种时间度量上的不一致现象.

在这种模式的实时语义中,我们使用了时态逻辑中的离散时间来表达各个事件(如赋值,通信就绪,通信发生等)的先后次序,称为时态次序. 同时也使用了时间变量(如 $t_{c!},t_{c?},t_s$ 等)来记录事件发生的实际时间. 在顺序进程的语义定义中,这两种时间度量总是一致的,即时态次序上的先后和实际时间的先后是无矛盾的. 在并发系统中采用了两个以上的时间变量 t_{s1},t_{s2},尽管在启动时要求它们取同样的值,但在进程的运行中,有可能各自记录各类事件发生的实际时间,而产生不一致性. 为避免此种不一致性,我们在顺序进程的语义中用 **until** 算子替代了一些"下一时刻"算子,即 O 算子. 以便使每个事件发生的时态时间(非实际时间)有很大的伸缩余地,从而可和实际时间保持一致.

时间变量的一致性可由下列公式表示

$$Con(S1,S2)=_{df}\ \square(((Ot_{s1}=r_1\wedge Ot_{s1}\neq t_{s1}\wedge\Diamond(Ot_{s2}=r_2\wedge Ot_{s2}\neq t_{s2}))\Rightarrow r_1\leqslant r_2)\wedge((Ot_{s2}=r_2\wedge Ot_{s2}\neq t_{s2}\wedge\Diamond(Ot_{s1}=r_1\wedge Ot_{s1}\neq t_{s2}))\Rightarrow r_2\leqslant r_1)).$$

$Con(S1,S2)$是由两部分组成. 第一部分说明,若 t_{s1} 记载了某一事件的发生,即 $Ot_{S1}\neq t_{S2}$,而且发生的实际时间为 r_1;那么,在其后 t_{s2} 记载的事件发生的实际时间应该不超前于 r_1,即 $r_1\leqslant r_2$. 第二部分是和第一部分对称的.

一旦规定了 SYN_c 和 $Con(S1,S2)$,以及通道时钟和进程时钟的一致性 SYN_{s1} 及 SYN_{s2},还有通信的同步要求 SYN,并发进程的语义就极为简单,退化为顺序进程的语义之合取式,

即

$$【s1 \parallel s2】 =_{df} 【s1】 \wedge 【s2】$$

这个语义定义具有所谓的组合性(Compositionality).

4 结束语

本文中讨论了一种模拟式的方法来刻划程序的实时语义,所模拟的环境因素是有选择的,但这种方法是有一般意义的.文中包含了很多复杂的公式,因为实时语义本身是极为复杂的.文献中一些简炼的语义,只是因为躲开了实时环境的很多复杂因素.实时语义的研究远非成熟,希望有兴趣的读者参加这项未尽的事业.

参考文献

1 Zhou Chaochen, Hansen M R, Ravn A P *et al*. Duration specifications for shared processors. LNCS 571, 1991. 21—32.

2 Pnueli A, Harel E. Applications of temporal logic to the specification of real time systems. LNCS 331, 1988. 84—98.

3 Hooman J. A denotational real time semantics for shared processors. LNCS 506, 1991. 185—201.

4 Reed G M, Roscoe A W. Metric spaces as models for real time concurrency. LNCS 298, 1987. 331—343.

5 Kaymans R. Specifying real time properties with metric temporal logic. Journal of Real Time Systems, 1990, (2).

6 Jahanian F, Mok A K—L. Safety analysis of timing properties in real time systems. IEEE Trans. SE, 1986, 12 (9): 890—904.

7 周巢尘.结构式时态语义.计算机应用与软件,1984,(1).

8 周巢尘.程序推理.计算机应用与软件.1984,(2).

DEFINING REAL TIME SEMANTICS BY SIMULATION

Han Yan

(*Institute of Software, The Chinese Academy of Sciences, Beijng* 100080)

Abstract The real time behavior of a program depends on its hard real time environment, such as the used compiler, scheduler, communication medium and computer. This paper is describing the real time behavior by simulating its environment, so that it provides a general approach to define real time semantics. The employed logic in the paper is the conventional linear temporal logic with explicit time variables.

Key words Real time semantics, timed CSP, temporal logic, simulation, scheduler.