ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.19, No.5, May 2008, pp.1134−1148
DOI: 10.3724/SP.J.1001.2008.01134

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

一种基于构件演算的主动构件精化方法*

陈 鑫[1,2+]

[1](南京大学 计算机科学与技术系,江苏 南京 210093)

[2](南京大学 计算机软件新技术国家重点实验室,江苏 南京 210093)

An Approach to Refining Active Components Based on Component Calculus

CHEN Xin[1,2+]

[1](Department of Computer Science and Technology, Nanjing University, Nanjing 210093, China)

[2](State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093, China)

+ Corresponding author: E-mail: chenxin@seg.nju.edu.cn, http://cs.nju.edu.cn

Chen X. An approach to refining active components based on component calculus. *Journal of Software*, 2008,19(5):1134−1148. http://www.jos.org.cn/1000-9825/19/1134.htm

Abstract: Modern component-based systems consist of active components that execute in parallel, which brings great difficulties in verifying correctness. By extending component calculus, a theory concerning refinement of active components is proposed. For interfaces, contracts are introduced which give functional specifications for both public methods and active action in terms of guarded designs. Then, a contract's dynamic behavior is defined by a pair of divergences/failures sets. The refinement relation between contracts is defined as the set inclusion of their divergences/failures sets. The theories applying simulation techniques to assure the refinement relation are proved. By defining the semantics of a component as a mapping from the contract of its required interface to the contract of its provide interface, component refinement can be proved in terms of contract refinement. When the component-based systems are being constructed in a bottom-up manner, the application of the refinement method together with the composition rule can guarantee their correctness.

Key words: interface; component; semantics; contract; refinement; composition

摘 要: 现代构件系统通常包含多个并发执行的主动构件,这使得验证构件系统的正确性变得十分困难.通过对构件演算进行扩展,提出了一种主动构件的精化方法.在构件接口层引入契约.契约使用卫式设计描述公共方法和主动活动的功能规约.通过一对发散、失败集合定义契约的动态行为,并利用发散、失败集合之间的包含关系定义契约间的精化关系.证明了应用仿真技术确认契约精化关系的定理.定义构件的语义为其需求接口契约到其服务接口契约的函数,以此为基础,可以通过契约的精化来证明构件的精化.给出了构件的组装规则.在构件系统自底向上的构造过程中,应用构件的精化方法和组装规则可以保证最终系统的正确性.

关键词: 接口;构件;语义;契约;精化;组合

* Supported by the National Basic Research Program of China under Grant No.2002CB312001 (国家重点基础研究发展计划(973)); the National High-Tech Research and Development Plan of China under Grant No.2007AA010302 (国家高技术研究发展计划(863)); the Natural Science Foundation of Jiangsu Province of China under Grant No.BK2007714 (江苏省自然科学基金)

Received 2007-11-15; Accepted 2008-02-19

中图法分类号: TP311 文献标识码: A

现代构件系统通常是由一组分布在广域网络上的主动构件构成的.这样,待整个系统开发完成并部署到运行环境之后再进行系统正确性的验证会变得非常困难.解决这个问题的途径之一就是研发由构造保证正确性(correct-by-construction)的理论,用其指导整个开发过程,从而确保整个系统的正确性.精化方法是一种实现由构造保证正确性的重要方法,它通过转换一步步地从规约构造出正确的最终实现.已有的精化理论仅支持从规约到结构化程序或是面向对象程序的转换[1,2].面向构件系统的精化理论和方法仍然是一个需要研究的问题.

主动构件是一种主动的软件实体,其内部有自主活动.什么时候执行这些自主活动,是由主动构件自身决定的,外部环境无法直接干预和控制这些活动的执行.在实现层面上,主动构件通常封装了一个进程或者是线程来驱动其内部的自主活动.通过主动构件提供的服务接口,环境可以获得主动构件提供的服务.主动构件服务方法和自主活动都可以调用需求接口中声明的方法.相应地,反应式构件是一种以被动方式工作的构件,它没有自主控制的内部活动,只是在服务接口上等待环境的调用,并在其服务方法的内部包含对需求接口中声明的方法的调用.可以从构件的外部行为对它们加以区分.反应式构件对需求方法的调用一定是在某个服务方法的执行过程中发生的,因此,反应式构件对需求方法的调用和返回事件总是被包含在环境对某个服务方法的调用事件和该服务方法的返回事件之间.同时,主动构件的服务接口也表现出非确定性,体现在一个对服务接口中方法的调用序列即使已在本次调用中被主动构件所接受,在下一次调用过程中,同样的调用序列仍有可能被主动构件所拒绝.如何为主动构件构造一个形式化模型,精确地刻画和分析主动构件的行为,并为主动构件的组装提供形式化支撑,是构件技术研究的重要内容.

构件演算以设计演算为基础,建立了反应式构件的形式化模型,研究了反应式构件的精化方法和组装方法.但是,构件演算没有提供直接描述主动构件自主行为的形式化机制,也未能给出分析自主行为如何影响主动构件交互行为的方法.因此,无法直接应用构件演算为主动构件建模.通过对构件演算进行扩展,本文给出了主动构件的行为模型,研究了主动构件的精化方法.首先,在接口的形式模型“契约”中加入对自主方法的描述.契约用卫式设计定义接口中公共方法和自主方法的功能规约.然后分析了自主方法会导致契约交互行为的非确定性,通过协议和一对发散、失败集合来描述契约的交互行为,给出了计算协议和发散、失败集合的方法.同时,用这对发散、失败集合上的包含关系定义契约的精化关系,证明了应用仿真技术判定契约精化关系的定理.构件的语义定义成一个映射其需求接口的契约到其服务接口的契约的函数.以此为基础,可以应用契约之间的精化关系来研究构件间的精化关系.通常,在构件系统中,构件的组装是通过需求接口和服务接口上方法的同步调用来实现的.本文给出了支持这种组装方法的形式规则.综合运用精化和组装的方法,可以在构件系统自底向上的过程中保证系统的正确性.

本文第 1 节介绍本文工作的基础——构件演算,并通过一个实例分析构件演算在分析主动构件时遇到的问题.第 2 节给出主动构件形式模型的定义,并以此为基础研究主动构件的精化方法和组装方法.第 3 节与相关工作做比较.第 4 节总结全文并对今后的工作进行展望.

1 构件演算

本文工作的形式化基础是构件演算.本节对构件演算作简要的介绍.下面的理论结果来自于文献[2–4].

不失一般性,设一种方法的型构为 $m(in:T_1,out:T_2)$,其中 m 是方法的名字,变量 in,out 表示输入、输出参数,类型 T_1,T_2 定义了输入、输出参数的类型.

本文使用文献[3]统一程序设计理论中的设计(design)作为语义定义的基础.设计将一个程序的执行描述为关于程序状态空间的一个关系.程序的状态空间定义在一组变量之上,这组变量记作α.程序的一个状态就是从这组变量到变量值空间的一个映射.

定义 1(设计). 一个定义在变量集合α上的设计 D 是一个二元组 $D=(\alpha,P)$,P 是一个作用于自由变量集合 $\alpha\cup\{ok,ok'\}$ 的谓词,P 定义为如下形式:

$$(p(in)\vdash R(in,out'))\overset{\text{def}}{=} ok\wedge p(in)\Rightarrow ok'\wedge R(in,out'),$$

其中:

- p 是一个与 $\{ok,ok'\}$ 无关的谓词,p 是前提条件,定义了初始状态.
- R 是一个与 $\{ok,ok'\}$ 无关的谓词,R 是后置条件,定义了结束状态.
- 布尔变量 ok 和 ok' 描述了程序正常开始和正常结束,它们不能被任何程序命令所操纵,无论是表达式还是赋值语句都不能改变它们的值.

设计规定,如果该设计被成功启动,即 ok=true,并且初始状态满足前置条件 p,那么它一定会终止,即 ok' = true,并且终止状态满足后置条件 R.

为了区分输出变量的起始和终止状态,对输出变量添加上标′表示其终止状态.

最弱前提条件的演算方法是分析程序性质的经典方法.定义 2 给出最弱前提条件和设计之间的关系.

定义 2(设计的最弱前提条件). $wp(p\vdash R,q)\overset{\text{def}}{=} p\wedge\neg(R;\neg q)$.

定义 3(设计的精化). 设计 $D_1=(\alpha,P_1)$ 被设计 $D_2=(\alpha,P_2)$ 精化,表示为 $D_1\sqsubseteq D_2$, 如果所有满足设计 D_2 的关系都满足设计 D_1,即 $[P_2\Rightarrow P_1]$,符号[]表示括号内的谓词对所有的情况都成立.

为了便于使用,在α明确的情况下,本文用省略形式 P 来代表设计 $D=(\alpha,P)$. 可用下面的定理来判定设计间的精化关系.

定理 1(精化关系判定). 两个设计 $P_i=p_i\vdash R_i(i=1,2)$ 之间有精化关系 $P_1\sqsubseteq P_2$ 当且仅当

$$[p_1\Rightarrow p_2]\wedge[(p_1\wedge R_2)\Rightarrow R_1].$$

文献[3]证明了设计对于程序构造算子是封闭的.

定理 2(设计的封闭性). 设计对于程序构造算子是封闭的.这里, $Cmd_1\sqcap Cmd_2$ 表示非确定选择结构,程序既可以选择执行 Cmd_1, 又可以选择执行 Cmd_2, 且选择的过程是程序外部无法干预的. $Cmd_1\lhd b\rhd Cmd_2$ 表示条件选择结构,条件 b 满足则执行 Cmd_1, 否则执行 Cmd_2.$Cmd_1;Cmd_2$.表示顺序组合算子,先执行 Cmd_1, 并以 Cmd_1 结束时的状态为 Cmd_2 的起始状态,紧接着执行 Cmd_2.

$$(p_1\vdash R_1)\sqcap(p_2\vdash R_2)\equiv(p_1\wedge p_2)\vdash(R_1\vee R_2),$$
$$(p_1\vdash R_1)\lhd b\rhd(p_2\vdash R_2)\equiv(p_1\lhd b\rhd p_2)\vdash(R_1\lhd b\rhd R_2),$$
$$(p_1\vdash R_1);(p_2\vdash R_2)\equiv(p_1\wedge\neg(R_1;\neg p_2))\vdash(R_1;R_2).$$

其中,谓词的顺序组合 $P(s');Q(s)$ 定义为 $\exists m\cdot P(m)\wedge Q(m)$.

本文后续证明过程需用到下面关于顺序组合算子 ; 的推论.

推论 1. 设 D_1 和 D_2 是设计,若 $\neg(D_1;D_2)[\text{true},\text{false}/ok,ok']$, 则 $(D_1;D_2)[\text{true},\text{true}/ok,ok']\equiv D_1[\text{true},\text{true}/ok,ok'];R_2$. 这里,符号 $[value_1,value_2/var_1,var_2]$ 表示分别用 $value_1,value_2$ 为变量 var_1,var_2 赋值.

对于一个反应式的程序,在两种情况下,可以观察到程序处于停机状态.可能是程序完成计算任务,正常停机;也可能是程序的计算任务进行到某点,需要与外界进行交互后,再继续执行下去,现正等待外界与之同步,处于等待状态.为了区分这两种状态,需要引入新的变量 $wait,wait'$. $[\text{true},\text{false}/ok',wait']$ 和 $[\text{true},\text{true}/ok',wait']$ 分别表示了上述两种状态.

可以利用设计定义反应式程序命令的语义,引入反应式设计.

定义 4(反应式设计). 一个设计 (α,P) 是一个反应式设计当且仅当 P 是映射 H 的一个不动点,这里,H 定义为 $H(P)\overset{\text{def}}{=}(\text{true}\vdash wait'\wedge_{v\in\alpha}v'=v)\lhd wait\rhd P$, 其中 $P_1\lhd b\rhd P_2$ 代表条件选择,当布尔表达式 b 为真时,执行设计 P_1, 否则执行设计 P_2.

考虑关于反应式设计顺序组合的一个健康条件.两个反应式设计的顺序组合 $P;Q$, 如果 Q 的初始状态是它的前驱 P 进入等待状态,那么 Q 保持当前状态不变,即 $Q=(\text{true}\vdash wait'\wedge v'=v)\lhd wait\rhd Q$.

在任意时刻,一个反应式构件中的方法并不都是可用的,构件只在可用的方法上与环境进行交互.如果外界环境试图调用一个不可用的方法,则由于环境和构件不能在交互的方法上达成一致,将导致它们进入相互等待

的状态.引入卫式条件来控制方法的可用性,方法是可用的当且仅当卫式条件为真.

定义 5(卫式设计). 一个卫式设计 $g \,\&\, D$ 是由布尔表达式 g 和设计 $D=(\alpha,P)$ 组成的,它定义为

$$g \,\&\, D \overset{\text{def}}{=} (\alpha, P \lhd g \rhd (\text{true} \vdash wait' \wedge_{v\in\alpha} v'=v)).$$

由定义 5 可知,如果设计 D 是一个反应式设计,那么卫式设计 $g \,\&\, D$ 也是一个反应式设计.

文献[4]用反应式设计定义程序命令的语义,并且证明了反应式设计对于程序构造算子也是封闭的,因而可以直接从方法的实现命令序列计算出一个反应式设计,并比较它是否与方法功能规约的设计部分之间有精化关系,从而检验方法的实现是否遵循规约.

关于设计、反应式设计和卫式设计的更多结果,可以参见文献[3,4].

构件演算通过契约层和构件层两个层次构造反应式构件的形式模型.在契约层,构件演算引入契约描述接口的行为,契约用卫式设计定义接口中方法的功能规约,用协议约束环境对接口方法的调用顺序.契约的动态交互行为定义在一对发散、失败集合上,集合中的基本事件是含有参数的函数调用事件和函数返回事件.用上述失败、发散集合的包含关系定义契约之间的精化关系.构件演算将构件的语义定义为从其需求接口契约到其服务接口契约的函数,因此,构件之间的精化关系可以在契约之间的精化关系上加以分析.

接口是构件和外界环境交互的设施,构件演算定义了如下标准形式的接口.

定义 6(接口). 一个接口 I 是对一组域变量和方法的声明.$I=\langle FDec, MDec\rangle$,其中:

- *FDec* 声明了一个域变量的集合,集合的每一个元素有如下的形式: $x:T$, x 和 T 分别是域变量的名称和类型.规定在同一个域变量集合的声明里,不同的域变量不能有相同的名字.使用 *I.FDec* 来引用接口中的域变量声明部分.
- *MDec* 声明了一种方法的集合,集合的每个元素是一种方法的型构(signature),形如 *m*(*inx*,*outy*)的型构声明了一种方法 *m*,它输入参数列表 *inx*,输出参数列表 *outy*.输入、输出参数列表中的每一项都有如下的形式:*u*:*U*,分别是参数的名称和类型.同样地,使用 *I.MDec* 来引用接口中方法声明的部分.

构件演算引入契约定义接口的语义.契约是一个四元组 $Ctr=\langle I, Init, MSpec, Prot\rangle$. I 是接口定义.*Init* 是一个设计,定义构件的初始化行为,*MSpec* 映射接口声明的每种方法到一个卫式设计,该卫式设计定义了方法的功能规约.*Prot* 是一组方法调用的序列,当环境按照这样的序列调用接口中的方法时,接口与环境不会发生死锁.

例 1:尝试用构件演算描述一个带缓冲的可计算输入参数平方值的主动构件 *Calculator*.构件 *Calculator* 接受输入值 x 并返回结果 $x*x$,它最多可以接受 10 个请求.调用 *Calculator* 的 *put* 方法提交请求,若缓冲仍有空余,请求被 *Calculator* 接受,放入请求集合,方法 *put* 立即返回,否则 *put* 被阻塞直到有空余为止.请求成功提交后,可调用 *get* 方法取得结果,若计算结果已存在于结果集合之中,则从结果集合中移去该计算结果,并将结果在方法 *get* 中返回,否则 *get* 被阻塞,一直等到查询的计算结果被放入结果集合为止.计算任务是由 *Calculator* 内部的自主方法 *Tau* 承担的,它随机地从请求集合中选取请求,计算其平方值,再将结果放入结果集合.

构件 *Calculator* 的服务接口定义如下:

$$\begin{aligned}
&REQUEST=\{id:INT,\ param:INT, result:INT\},\\
&I.FDec=\{requests:[REQUEST], results:[REQUEST]\},\\
&I.MDec=\{put(in\ x:INT, out\ \ y:INT), get(in\ id:INT, out\ y:INT)\}.
\end{aligned}$$

用契约描述上述接口的语义:

$$\begin{aligned}
&Init = requests'=\varnothing \wedge results'=\varnothing\\
&MSpec(put) = |requests|+|results|<10\ \&\\
&\qquad \text{true} \vdash \exists req:REQUEST\cdot req.id'\notin idset \wedge req.param=x \wedge y'=req.id' \wedge\\
&\qquad\qquad requests'=requests\cup\{req\} \wedge results'=results\\
&MSpec(get) = \exists req:REQUEST\cdot req.id=id \wedge req\in results\ \&\\
&\qquad \text{true} \vdash \exists req:REQUEST\cdot req.id=id \wedge req\in results \wedge y'=req.result \wedge\\
&\qquad\qquad requests'=requests \wedge results'=results\setminus\{req\}
\end{aligned}$$

其中,接口 I 定义了类型 $REQUEST$,$[type]$表示一个元素类型为 $type$ 的集合,$|X|$返回集合 X 中元素的个数,$idset=\{req.id:INT \mid req\in requests\vee req\in results\}$.

构件演算以契约中方法的功能规约为基础,计算出契约的协议和发散、失败集合.

利用这样的计算方法,确定契约的协议:

$$\forall\langle ?m_1(x_1),...?m_k(x_k)\rangle\in Prot\cdot wp(Init;g_{m_1}\ \&\ D_{m_1}[x_1/in_1];...;g_{m_k}\ \&\ D_{m_k}[x_k/in_k],\neg wait\wedge\exists m\in MDec\cdot g_m)=\text{true}.$$

对于例 1,$\langle ?put(3,id),?get(id,9)\rangle$ 是 $Calculator$ 服务接口所允许的一次交互,应该出现在契约的协议中,然而

$$wp(Init;g_{put}\ \&\ D_{put}[3,id/x,y];g_{get}\ \&\ D_{get}[id/x],\neg wait)\neq\text{true}.$$

同样地,考虑契约失败集合中的一种情况,在环境与构件按照序列 $\langle ?m_1(x_1),m_1(y_1)!,...,?m_k(x_k),m_k(y_k)!\rangle$ 进行交互之后,构件拒绝响应集合 X 中的方法调用事件,因为此时这些方法的卫式条件为假.

$$\left\{\begin{array}{l}(\langle ?m_1(x_1),m_1(y_1)!,...,?m_k(x_k),m_k(y_k)!\rangle,X)\mid\\ \exists v'\cdot\left(Init;g_{m_1}\ \&\ D_{m_1}[x_1,y_1/in_1,out_1'];\right.\\ ...;\\ \left.g_{m_k}\ \&\ D_{m_k}[x_k,y_k/in_k,out_k']\right)[\text{true},\text{false},\text{true},\text{false}/ok,wait,ok',wait']\wedge\forall ?m\in X\cdot\neg g_m[v'/v]\end{array}\right\}$$

使用上述计算方法,可有如下结果:$(\langle Init,?put(3),put(id)!\rangle,\{?get,get!\})$ 属于契约的失败集合.根据例 1 中关于构件功能的描述,这一结论是不正确的.

发散、失败集合是构件演算研究契约精化关系的基础,发散、失败集合的包含关系定义了契约间的精化关系,而构件的精化关系是定义在契约精化关系之上的.对于主动构件的接口,上述计算方法已经失效,它导致无法直接应用构件演算分析主动构件的交互行为和精化关系.

在研究主动构件的形式化模型之前,有必要将主动构件的自主行为与反应式构件私有方法的行为加以区分.反应式构件的私有方法是通过将服务接口中的公有方法加以掩藏后得到的.私有方法不能被反应式构件的外部环境所调用,作为一种被动的软件实体,反应式构件自身也不会主动地调用这些私有方法.若接口 I 对应契约 Ctr,隐藏 I 中所有在集合 S 中出现的方法,得到一个新的接口 $I\backslash S$,接口 $I\backslash S$ 对应契约 $Ctr\backslash S$.文献[4]指出,契约 $Ctr\backslash S$ 的失败集合可以从契约 Ctr 中得到:

$$F(Ctr\backslash S)\overset{\text{def}}{=}\{(s,X)\mid(s,X)\in F(Ctr)\wedge s\in\{?m(x),m(y)!\mid m\in MDec\backslash S\}^*\wedge X\in\{?m,m!\mid m\in MDec\backslash S\}\},$$

其中,失败集合中的序列 s 是接口与环境可能的交互序列.上述定义表明,当公共方法被隐藏而成为私有方法后,接口可能的交互序列就是在原有的交互序列中选出那些不包含私有方法的序列.这意味着,反应式构件不会主动调用这些私有方法,从而使原接口中包含私有方法的那些交互序列仍然有效.主动构件的自主方法是由构件自身启动并执行的方法,它不受构件外部环境的直接控制,与反应式构件的私有方法有着本质上的不同.

2 主动构件的形式模型和精化方法

由上一节的分析可知,在为主动构件的行为建模时,需要解决两个根本问题:一是如何建立构件自主行为的直接描述机制,二是在为构件交互行为建模时,如何分析自主行为对构件交互行为的影响.

本节从解决这两个问题入手来研究主动构件的形式模型和精化方法.和构件演算一样,分接口层和构件层来进行讨论.在接口层,对契约进行扩展,用卫式设计直接定义自主活动的功能规约.自主活动对构件交互行为的影响体现在计算契约的协议和发散、失败集合的过程中.因此,需要重新定义它们的计算方法.重新研究了用仿真技术证明契约精化关系的定理,并基于新的计算契约发散、失败集合的方法证明了该定理.在构件层,通过将构件定义为从其需求接口的契约到其服务接口的契约的一个映射,可以应用契约精化关系的证明定理来研究构件的精化关系.另外,在构件的语义定义中,构件的需求接口输入一个自主契约后,复合构件自主活动的功能规约是用非确定选择符⊓对两个构件自主活动的功能规约进行组合.

2.1 接口的形式化模型和精化方法

除了方法声明部分说明的方法以外,接口还包含了两类特殊的方法:初始化方法和内部自主执行的方法.在构件启动时刻,立即启动初始化方法为接口中的每个域变量赋初值.作为一个主动的实体,通常构件还定义了一系列自主的活动,当某种条件满足时,构件自动执行这些活动.这两类方法的活动,都会对接口的整体行为产生影响,因而,在定义接口的语义时,必须包含对这两类活动的描述.引入契约定义接口的语义.

定义 7(契约). 一个契约是一个多元组 $Ctr=\langle I,Init,Tau,MSpec,Prot\rangle$,其中:

- I 是一个接口.
- $Init$ 是一个设计,定义接口的初始化活动,型如 $\text{true}\vdash init(I.FDec')\wedge\neg wait$,其中 $init$ 是一个谓词.
- Tau 是一个卫式设计 $(I.FDec\cup I.FDec', g_T \,\&\, D_T)$,它定义了构件自主的内部行为,即当卫式条件 g_T 为真时,构件可以自动地执行设计 D_T 定义的行为.
- $MSpec$ 为每种方法 $m(x:U,y:V)$ 指定了一个卫式设计 $(\alpha_m, g_m \,\&\, D_m)$,这里,$\alpha_m$ 是设计的字符表,由接口的域变量和方法的输入、输出参数组成. $\alpha_m \overset{\text{def}}{=} \{x\}\cup I.FDec\cup\{y'\}\cup I.FDec'$.
- $Prot$ 称为协议,是一组方法调用序列的集合.集合里的每个元素是一个形如 $?m_1(x_1),...?m_k(x_k)$ 的序列,$?m_i(x_i)$ 表示对 $I.MDec$ 中声明的方法 m_i 的一个调用,输入参数是 x_i.

契约的协议是由契约中各种方法的卫式设计决定的.给定一个契约,可以计算它的协议部分是否与契约的方法规约部分相一致.

定义 8(契约一致性). 称一个契约 $Ctr=\langle I,Init,Tau,MSpec,Prot\rangle$ 是一致的,如果它满足以下条件:

$$\forall\langle ?m_1(x_1),...?m_k(x_k)\rangle\in Prot\cdot$$

$$\exists n_1,n_2,...,n_{k+1}\cdot wp\left(Init;\tau^{n_1};g_{m_1}\,\&\,D_{m_1}[x_1/in_1];...\tau^{n_k};g_{m_k}\,\&\,D_{m_k}[x_k/in_k];\tau^{n_{k+1}},\neg wait\wedge\exists m\in MDec\cdot g_m\right)=\text{true}\wedge$$

$$n_1\geq 0\wedge n_1<\infty\wedge...n_{k+1}\geq 0\wedge n_{k+1}<\infty.$$

其中,$\tau^{n_i}=\underbrace{g_T\,\&\,D_T;...g_T\,\&\,D_T}_{n_i}, 0\leq n_i<\infty$,表示构件自动执行了 n_i 次 Tau 所对应的卫式设计.

为了便于书写,下文中使用 $(g_T\,\&\,D_T)^{n_i}$ 表示 $\underbrace{g_T\,\&\,D_T;...g_T\,\&\,D_T}_{n_i}$.

定义 8 提供了从契约中的 $MSpec$ 计算 $Prot$ 的方法.不难发现,对于契约 $Ctr=(I,Init,Tau,MSpec)$,存在不止一个 $Prot$ 与其相一致,我们把包含所有一致交互序列的集合称为最弱一致协议,记作 $WProt$.

定义 9(最弱一致协议). 一个契约 $Ctr=\langle I,Init,Tau,MSpec\rangle$ 的最弱一致协议 $WProt$ 定义为

$$WProt\overset{\text{def}}{=}\left\{\begin{array}{l}\langle ?m_1(x_1),...,?m_k(x_k)\rangle\mid\\ \exists n_1,n_2,...,n_{k+1}\cdot n_1\geq 0\wedge n_1<\infty\wedge...\wedge n_{k+1}\geq 0\wedge n_{k+1}<\infty\wedge\\ \quad wp(Init;\tau^{n_1};g_{m_1}\,\&\,D_{m_1}[x_1/x];...\tau^{n_k};g_{m_k}\,\&\,D_{m_k}[x_k/x];\tau^{n_{k+1}},\neg wait\wedge\exists m\in MDec\cdot g_m)=\text{true}\end{array}\right\}.$$

其中,$\tau^{n_i}=(g_T\,\&\,D_T)^{n_i}, 0\leq n_i<\infty$.

在前一种方法返回以后,主动构件可以响应外部对一个卫式命令为真的方法的调用,也可以自动执行 Tau 方法,若 Tau 方法的卫式条件 g_T 为真.这一选择过程是构件随机独立完成的,外部无法干预.可能出现这样一种情况,即 g_T 总是真,且构件总是选择自动执行 Tau,这时,构件不响应任何外部请求,一直在运转不停机,进入发散状态.同时,也不允许 Tau 活动在几次连续的执行过程中自行发散.为排除这些病态情况,定义契约的健康条件:一个健康契约的 Tau 最多可以连续执行 k 次且在连续执行 k 次之前不会发散,即 $\exists k:N\cdot(wp((g_T\,\&\,D_T)^k,wait')=\text{false}\wedge k<\infty\wedge\left(\forall j<k\cdot wp((g_T\,\&\,D_T)^j,ok')=\text{true}\right)\right)$.

根据上述关于主动构件行为的描述可知,主动构件的最弱一致协议与反应式构件的最弱一致协议有着不同的语义.反应式构件最弱一致协议中包含的协议之间是确定性的选择关系.环境一旦选取某个协议与构件进行交互,最后的交互场景一定遵循该协议.而对于主动构件,其最弱一致协议中包含的协议之间是一种非确定性

的选择关系.也就是说,反应式构件只是提供了一组可能的交互协议,环境与构件交互行为的实际结果会受到外部环境选择行为和构件内部随机选择行为的共同影响.考虑例 1 中构件 *Calculator* 接收请求序列 $\langle ?put(3,id_1),?put(2,id_2)\rangle$ 之后的行为,由于构件随机决定处理请求的顺序,因此序列 $\langle ?put(3,id_1),?put(2,id_2),?get(id_1,9),?get(id_2,4)\rangle$ 和 $\langle ?put(3,id_1),?put(2,id_2),?get(id_2,4),?get(id_1,9)\rangle$ 都是可能的交互结果,但事先无法确定构件会按照哪个序列与环境进行交互.

例 2:考虑例 1 中构件服务接口的契约.它的服务接口 I 有如下定义:

$$REQUEST=\{id:INT,\ param:INT,\ result:INT\},$$
$$I.FDec=\{requests:[REQUEST],\ results:[REQUEST]\},$$
$$I.MDec=\{put(in\ x:INT,\ out\ \ y:INT),\ get(in\ id:INT,\ out\ y:INT)\}.$$

引入契约定义该接口的规约.

$$Init=requests'=\varnothing\wedge results'=\varnothing,$$
$$Tau=requests\neq\varnothing\ \&$$
$$\text{true}\vdash\exists req:REQUEST\cdot req\in requests\wedge req.result'=req.param*req.param\wedge$$
$$req.id'=req.id\wedge requests'=requests\setminus\{req\}\wedge results'=results\cup\{req\},$$
$$MSpec(put)=|requests|+|results|<10\ \&$$
$$\text{true}\vdash\exists req:REQUEST\cdot req.id'\notin idset\wedge req.param=x\wedge$$
$$y'=req.id'\wedge requests'=requests\cup\{req\}\wedge results'=results,$$
$$MSpec(get)=\exists req:REQUEST\cdot req.id=id\wedge req\in results\ \&$$
$$\text{true}\vdash\exists req:REQUEST\cdot req.id=id\wedge req\in results\wedge$$
$$y'=req.result\wedge requests'=requests\wedge results'=results\setminus\{req\}.$$

其中, $idset=\{req.id:INT\mid req\in requests\vee req\in results\}$, $[type]$ 表示一个元素类型为 *type* 的集合, $|X|$ 返回集合 X 中元素的个数.

契约构造了接口的一个状态模型,状态空间是由接口中所有的域变量和所有方法的输入、输出参数加上特殊变量 $ok,ok',wait,wait'$ 构成的.在比较两个接口对应方法之间的功能规约时,应用状态模型比较方便.若研究多个接口之间复杂的交互行为,则利用基于事件的模型更加方便.考虑为契约的动态行为建立一个基于事件的模型.从外部来看,接口与外界的交互行为表现为一系列带参数的方法调用和返回.若接口工作正常,则方法调用事件的直接后继事件必定是该方法的返回事件;如果接口死锁了,则在某个方法调用事件之后,观察不到任何事件;如果接口发散,则方法的调用事件的后继事件可以是任意的事件,而不一定是该方法的返回事件.为了区分处理上述 3 种情况,本文将带输入参数的方法调用事件和带输出参数的方法返回事件作为独立的事件分开处理.据此,将契约的动态行为定义在包含参数的方法调用和方法返回事件序列之上.

定义 10(契约的动态行为). 契约的动态行为是一个三元组 $(Trace,F(Ctr),D(Ctr))$, 其中:

- 集合 $D(Ctr)$ 是一组契约与其外部环境交互的序列,这样的交互会导致契约发散.

$$D(Ctr)\overset{\text{def}}{=}\left\{\begin{array}{l}\langle ?m_1(x_1),m_1(y_1)!,\ldots ?m_k(x_k),m_k(y_k)!,?m_{k+1}(x_{k+1})\rangle\cdot s\mid\\ \exists v,v',wait',n_1,\ldots,n_{k+1}\cdot\left(Init;\tau^{n_1};g_{m_1}\ \&\ D_{m_1}[x_1,y_1/x,y'];\ldots\tau^{n_k};\right.\\ \left.g_{m_k}\ \&\ D_{m_k}[x_k/x];\tau^{n_{k+1}}\right)[\text{true},\text{false}/ok,ok']\end{array}\right\},$$

其中, $\tau^n=(g_T\ \&\ D_T)^n(0\le n<\infty)$ 是构件自动连续执行 n 次 *Tau* 对应的卫式设计. $?m_i(x_i)$ 是对方法 m_i 的调用事件,x_i 是输入的实参, $m_i(y_i)!$ 是方法 m_i 执行结束以后返回并带回输出参数 y_i 的事件,x 和 y 是方法 m_i 的输入、输出形参. $g_{m_i}\ \&\ D_{m_i}$ 是方法 m_i 对应的卫式设计,v 和 v' 记录了方法执行前后域变量的值.

- $F(Ctr)$是二元组 (s,X) 的集合,当构件和外界环境按照 s 中定义的序列发生交互以后,构件拒绝参与执行集合 X 中包含的事件.

$$succ \overset{\text{def}}{=} (\text{true},\text{false},\text{true},\text{false})/ok,wait,ok',wait'$$

$$rej \overset{\text{def}}{=} (\text{true},\text{false},\text{true},\text{true})/ok,wait,ok',wait'$$

$$F(Ctr) \overset{\text{def}}{=} \left\{(\langle\rangle, X) \mid \exists v', n_1 \cdot (Init;\tau^{n_1})[succ] \wedge \forall ?m \in X \cdot \neg g_m[v'/v]\right\} \cup$$

$$\left\{\begin{array}{l}(\langle ?m_1(x_1), m_1(y_1)!, ..., ?m_k(x_k), m_k(y_k)!\rangle, X) \mid \\ \exists v', n_1, ..., n_{k+1} \cdot \left(Init;\tau^{n_1};g_{m_1} \,\&\, D_{m_1}[x_1,y_1/in_1,out_1'];\right. \\ ...; \\ \left.\tau^{n_k};g_{mk} \,\&\, D_{mk}[x_k,y_k/in_k,out_k']\right);\tau^{n_{k+1}}[succ] \wedge \forall ?m \in X \cdot \neg g_m[v'/v]\end{array}\right\} \cup$$

$$\left\{\begin{array}{l}(\langle ?m_1(x_1), m_1(y_1)!, ..., ?m_k(x_k)\rangle, X) \mid \\ \exists v', n_1, ..., n_k \cdot \left(Init;\tau^{n_1};g_{m1} \,\&\, D_{m1}[x_1,y_1/in_1,out_1'];\right. \\ ...; \\ \left.\tau^{n_k};g_{mk} \,\&\, D_{mk}[x_k,y_k/in_k,out_k']\right)[succ] \wedge m_k !\notin X\end{array}\right\} \cup$$

$$\left\{\begin{array}{l}(\langle ?m_1(x_1), m_1(y_1)!, ..., ?m_k(x_k)\rangle, X) \mid \\ \exists v', n_1, ..., n_k \cdot \left(Init;\tau^{n_1};g_{m_1} \,\&\, D_{m_1}[x_1,y_1/in_1,out_1'];\right. \\ ...; \\ \left.\tau^{n_{k-1}};g_{m_{k-1}} \,\&\, D_{m_{k-1}}[x_{k-1},y_{k-1}/in_{k-1},out_{k-1}']\right)[succ];\tau^{n_k};g_{m_k} \,\&\, D_{m_k}[x_k/in_k][rej]\end{array}\right\} \cup$$

$$\{(s,X) \mid s \in D(Ctr)\}.$$

$F(Ctr)$定义了 5 种情况,当环境调用集合 X 中包含的方法时,由于构件拒绝接收,导致系统发生死锁.它们分别是:

(1) 第 1 个子集表示系统初始化以后,集合 X 中包含的方法调用事件被拒绝,因为它们的卫式条件为假;

(2) 第 2 个子集表示在系统发生了 s 定义的事件序列以后,集合 X 中包含的方法调用事件被拒绝,因为它们的卫式条件为假;

(3) 第 3 个子集表示调用方法 m_k 后,正在等待它返回,因而拒绝除方法 m_k 返回事件以外的所有事件;

(4) 第 4 个子集表示调用方法 m_k 后,进入死锁状态;

(5) 第 5 个子集表示导致系统发散的调用序列,也被系统所拒绝.

定义契约的 *Trace* 为所有在 $F(Ctr)$中出现的序列:

$$Trace(Ctr) \overset{\text{def}}{=} \{s \mid \exists X \cdot (s,X) \in F(Ctr)\}.$$

定义 10 给出了从契约计算出契约的一对发散、失败集合的方法,这样的计算过程保证了两者的一致性.

基于契约的动态行为,可以定义契约间的精化关系.

定义 11(契约精化). 契约 Ctr_1 被契约 Ctr_2 精化,表示为 $Ctr_1 \sqsubseteq_{ctr} Ctr_2$,如果:

(1) Ctr_2 提供了与 Ctr_1 同样的方法,即 $Ctr_1.MDec = Ctr_2.MDec$;

(2) 相对于 Ctr_1 来说,Ctr_2 更加不容易发散,即 $D(Ctr_1) \supseteq D(Ctr_2)$;

(3) 相对于 Ctr_1 来说,Ctr_2 更加不容易死锁,即 $F(Ctr_1) \supseteq F(Ctr_2)$.

定义 12(契约等价). 如果契约 Ctr_1 和 Ctr_2 之间有相互精化的关系,则称契约 Ctr_1 和 Ctr_2 等价,表示为 $Ctr_1 \equiv Ctr_2$.

可以直接比较两个契约对应的发散、失败集合之间的包含关系,以确定契约的精化关系.通常发散、失败集合本身会非常的庞大,尽管可以利用工具如 FDR[5]来辅助比较的过程,但这仍然是一个比较繁琐的工作.简化证明过程的一种方法是以契约的状态模型为基础,使用仿真技术来证明契约间的精化关系.下面的两个定理给出了利用仿真技术证明契约精化关系的方法.

定理 3. 向下仿真蕴含精化.设两个契约 $Ctr^i = (I^i, Init^i, Tau^i, MSpec^i)$ 的接口有相同的方法声明,即

$I^1.MDec = I^2.MDec$, 它们之间有精化关系 $Ctr^1 \sqsubseteq_{ctr} Ctr^2$, 如果存在一个从 $Ctr^1.FDec$ 到 $Ctr^2.FDec$ 的满映射 ρ,满足以下条件,则称 ρ 是从 Ctr_1 到 Ctr_2 向下的仿真:

(1) ρ保持初始条件: $\exists n_1, n_2 \cdot Init^2;(g_T^2 \wedge D_T^2)^{n_2} \Rightarrow (Init^1;(g_T^1 \wedge D_T^1)^{n_1};\rho) \wedge\ 0 \le n_1, n_2 < \infty$.

(2) 对接口中所有的方法 m, ρ 保持卫式条件的等价: $[\rho \Rightarrow g_m^1 = g_m^2]$, 其中 $MSpec^i(m) = g_m^i \ \&\ D_m^i, i = 1,2$.

(3) 对于接口中每种方法 m 对应的卫式设计有:

$$\exists n_{1m}, n_{2m} \cdot (g_m^1 \wedge D_m^1);(g_T^1 \wedge D_T^1)^{n_{1m}};sim \sqsubseteq sim;(g_m^2 \wedge D_m^2);(g_T^2 \wedge D_T^2)^{n_{2m}} \wedge 0 \le n_{1m}, n_{2m} < \infty,$$

其中, $MSpec^i(m) = g_m^i \ \&\ D_m^i, i = 1,2, sim \overset{\text{def}}{=} \text{true} \vdash (\rho(u,v') \wedge wait' = wait)$.

证明:由条件(1)可有推论: (a) $(Init^1;(g_T^1 \ \&\ D_T^1)^{n_1};\rho) \sqsubseteq Init^2;(g_T^2 \ \&\ D_T^2)^{n_2}, 0 \le n_1, n_2 < \infty$.

由条件(2)、(3)有如下推论: (b) $(Init^1;TR^1;sim) \sqsubseteq (Init^2;TR^2)$, 其中,

$$TR^i = (g_T^i \ \&\ D_T^i)^{n_i};(g_{m1}^i \ \&\ D_{m1}^i);(g_T^i \ \&\ D_T^i)^{n_{im1}} ...;(g_{mk}^i \ \&\ D_{mk}^i);(g_T^i \ \&\ D_T^i)^{n_{imk}}.$$

为叙述方便,定义如下常量:

$$DIV \overset{\text{def}}{=} \text{true}, \text{false} / ok, ok',$$

$$SUCC \overset{\text{def}}{=} \text{true}, \text{false}, \text{true}, \text{false} / ok, wait, ok', wait'.$$

首先证明(a):

$\langle ?m_1(x_1), m_1(y_1)!, ...?m_k(x_k), m_k(y_k)!, ?m_{k+1}(x_{k+1})\rangle \cdot s \in D(Ctr^2)$

{Definition of Divergence}

$\Rightarrow \exists v, v', wait' \cdot (Init^2;(g_T^2 \ \&\ D_T^2)^{n_2};(g_{m1}^2 \ \&\ D_{m1}^2)[x_1, y_1 / x, y'];...;(g_{mk}^2 \ \&\ D_{mk}^2);(g_T^2 \ \&\ D_T^2)^{n_{2mk}}[x_k / x])[DIV]$

$\{(g \ \&\ D)[\text{false} / wait'] = (g \wedge D)[\text{false} / wait']\}$

$\Rightarrow \exists v, v', wait' \cdot (Init^2;(g_T^2 \wedge D_T^2)^{n_2};(g_{m1}^2 \wedge D_{m1}^2)[x_1, y_1 / x, y'];...;(g_{mk}^2 \wedge D_{mk}^2);(g_T^2 \wedge D_T^2)^{n_{2mk}}[x_k / x])[DIV]$

{Conclusion (b)}

$\Rightarrow \exists u, u', wait' \cdot (Init^1;(g_T^1 \wedge D_T^1)^{n_1};(g_{m1}^1 \wedge D_{m1}^1)[x_1, y_1 / x, y'];...;(g_{mk}^1 \wedge D_{mk}^1);(g_T^1 \wedge D_T^1)^{n_{1mk}}[x_k / x];sim)[DIV]$

{Definition of sim}

$\Rightarrow \exists u, u', wait' \cdot (Init^1;(g_T^1 \wedge D_T^1)^{n_1};(g_{m1}^1 \wedge D_{m1}^1)[x_1, y_1 / x, y'];...;(g_{mk}^1 \wedge D_{mk}^1);(g_T^1 \wedge D_T^1)^{n_{1mk}}[x_k / x])[DIV]$

$\{(g \ \&\ D)[\text{false} / wait'] = (g \wedge D)[\text{false} / wait']\}$

$\Rightarrow \exists u, u', wait' \cdot (Init^1;(g_T^1 \ \&\ D_T^1)^{n_1};(g_{m1}^1 \ \&\ D_{m1}^1)[x_1, y_1 / x, y'];...;(g_{mk}^1 \ \&\ D_{mk}^1);(g_T^1 \ \&\ D_T^1)^{n_{1mk}}[x_k / x])[DIV]$

{Definition of Divergence}

$\Rightarrow \langle ?m_1(x_1), m_1(y_1)!, ...?m_k(x_k), m_k(y_k)!, ?m_{k+1}(x_{k+1})\rangle \cdot s \in D(Ctr^1)$

然后证明 $F(Ctr^2) \subseteq F(Ctr^1)$,需要考虑 3 种情况:

情况 1:

$(\langle\rangle, X) \in F(Ctr^2)$

{Definition of Failures}

$\Rightarrow \exists v' \cdot (Init^2;(g_T^2 \wedge D_T^2)^{n_2})[\text{true}, \text{false}, \text{true}, \text{false} / ok, wait, ok', wait'] \wedge \forall ?m \in X \cdot \neg g_m^2[v' / v])$

{Conclusion (a)}

$\Rightarrow \exists v', u' \cdot (Init^1;(g_T^1 \wedge D_T^1)^{n_1})[\text{true}, \text{false}, \text{true}, \text{false} / ok, wait, ok', wait'] \wedge \rho(u', v') \wedge \forall ?m \in X \cdot \neg g_m^2[v' / v])$

{Condition (2)}

$\Rightarrow \exists u' \cdot (Init^1;(g_T^1 \wedge D_T^1)^{n_1})[\text{true}, \text{false}, \text{true}, \text{false} / ok, wait, ok', wait'] \wedge \forall ?m \in X \cdot \neg g_m^1(u'))$

{Definition of Failures}

$\Rightarrow (\langle\rangle, X) \in F(Ctr^1)$

情况 2:

$(\langle ?m_1(x_1), m_1(y_1)!, ..., ?m_k(x_k), m_k(y_k)!\rangle, X) \in F(Ctr^2)$

{Definition of Failures and $(g \,\&\, D)[\text{false}/wait'] = (g \wedge D)[\text{false}/wait']$}

$$\Rightarrow \exists v' \cdot \left(Init^2; (g_T^2 \wedge D_T^2)^{n_2}; (g_{m1}^2 \wedge D_{m1}^2)[x_1, y_1 / x, y']; ...; (g_{mk}^2 \wedge D_{mk}^2)[x_k, y_k / x, y']; (g_T^2 \wedge D_T^2)^{n_{2mk}} \right)[SUCC] \wedge \forall ?m \in X \cdot \neg g_m^2[v'/v])$$

{Conclusion (b) and corollary 1}

$$\Rightarrow \exists u', v' \cdot \left(Init^1; (g_T^1 \wedge D_T^1)^{n_1}; (g_{m1}^1 \wedge D_{m1}^1)[x_1, y_1 / x, y']; ...; (g_{mk}^1 \wedge D_{mk}^1)[x_k, y_k / x, y']; (g_T^1 \wedge D_T^1)^{n_{1mk}} \right)[SUCC] \wedge \rho(u', v') \wedge \forall ?m \in X \cdot \neg g_m^2[v'/v]$$

{Condition 2}

$$\Rightarrow \exists u' \cdot \left(Init^1; (g_T^1 \wedge D_T^1)^{n_1}; (g_{m1}^1 \wedge D_{m1}^1)(x_1, y_1 / x, y'); ...; (g_{mk}^1 \wedge D_{mk}^1)[x_k, y_k / x, y']; (g_T^1 \wedge D_T^1)^{n_{1mk}} \right)[SUCC] \wedge \forall ?m \in X \cdot \neg g_m^1(u')$$

{Definition of Failures and $(g \,\&\, D)[\text{false}/wait'] = (g \wedge D)[\text{false}/wait']$}

$$\Rightarrow (\langle ?m_1(x_1), m_1(y_1)!, ..., ?m_k(x_k), m_k(y_k)! \rangle, X) \in F(Ctr^1)$$

情况 3:

$$(\langle ?m_1(x_1), m_1(y_1)!, ..., ?m_k(x_k) \rangle, X) \in F(Ctr^2)$$

{Definition of Failures and $(g \,\&\, D)[\text{false}/wait'] = (g \wedge D)[\text{false}/wait']$}

$$\Rightarrow \exists v', y' \cdot \left(Init^2; (g_T^2 \wedge D_T^2)^{n_2}; (g_{m1}^2 \wedge D_{m1}^2)[x_1, y_1 / x, y']; ...; (g_{mk}^2 \wedge D_{mk}^2)[x_k / x] \right)[SUCC]) \wedge m_k ! \notin X$$

{Conclusion (b) and corollary 1}

$$\Rightarrow \exists u', v', y' \cdot \left(Init^1; (g_T^1 \wedge D_T^1)^{n_{11}}; (g_{m1}^1 \wedge D_{m1}^1)[x_1, y_1 / x, y']; ...; (g_{mk}^1 \wedge D_{mk}^1)[x_k / x] \right)[SUCC]) \wedge \rho(u', v') \wedge m_k ! \notin X$$

{Definition of Failures and $(g \,\&\, D)[\text{false}/wait'] = (g \wedge D)[\text{false}/wait']$}

$$\Rightarrow (\langle ?m_1(x_1), m_1(y_1)!, ..., ?m_k(x_k) \rangle, X) \in F(Ctr^1)$$

由结论 $D(Ctr^2) \subseteq D(Ctr^1)$ 和结论 $F(Ctr^2) \subseteq F(Ctr^1)$ 可知 $Ctr^1 \sqsubseteq Ctr^2$. □

例 3:考虑如下两个契约:

契约 1:

$$I.FDec = buf : INT^*$$
$$I.MDec = \{put(in\ x : INT), get(out\ y : INT)\}$$
$$Init = buf' = \langle\rangle$$
$$Tau = \text{false} \,\&\, skip$$
$$MSpec(put) = |buf| \le 1 \,\&\, \text{true} \vdash buf' = buf^\frown \langle x \rangle$$
$$MSpec(get) = |buf| > 0 \,\&\, \text{true} \vdash buf' = tail(buf) \wedge y' = head(buf)$$

其中,INT^*表示一个整数序列,设计 $skip$ 是这样的一个设计: $skip \overset{\text{def}}{=} \text{true} \vdash \wedge_{x \in in\alpha} x' = x$, 它保持字符表中所有变量的值不变. $|s|$ 返回序列 s 中元素的个数, $s_1^\frown s_2$ 表示连接序列 s_2 的首部与序列 s_2 的尾部成为一个序列, $head(s), tail(s)$ 分别返回序列 s 头部的第 1 个元素和序列 s 除去头部的第 1 个元素之后的序列.

契约 2:

$$I.FDec = buf_1 : [INT], buf_2 : [INT]$$
$$I.MDec = \{put(in\ x : INT), get(out\ y : INT)\}$$
$$Init = buf_1' = \varnothing \wedge buf_2' = \varnothing$$
$$Tau = buf_1 = \varnothing \,\&\, \text{true} \vdash buf_1' = buf_2 \wedge buf_2' = \varnothing$$
$$MSpec(put) = buf_2 = \varnothing \,\&\, \text{true} \vdash buf_2' = \{x\}$$
$$MSpec(get) = buf_1 \neq \varnothing \,\&\, \text{true} \vdash buf_1' = \varnothing \wedge \{y'\} = buf_1$$

应用定理 3 可以证明契约 2 是契约 1 的精化,它们都定义了一个容量为 2 的缓冲.

定理 3 并不直接比较自主活动 Tau 之间是否有精化关系,而是将自主活动和紧邻其前发生的服务方法加以

组合后,再比较组合后服务方法之间的精化关系.事实上,由于接口的主动活动对外部环境是不可见的,因而这些内部活动很容易被外部认为是与之相邻发生的前一种服务方法的一部分.

定理 4. 向上仿真蕴含精化.两个契约 $Ctr^i=(I^i,Init^i,Tau^i,MSpec^i)$ 的接口有相同方法声明,即 $I^1.MDec=I^2.MDec$,它们之间有精化关系 $Ctr^1\sqsubseteq_{ctr}Ctr^2$,如果存在一个从 $Ctr^2.FDec$ 到 $Ctr^1.FDec$ 的满映射 $\rho(v,u')$,满足以下条件,则称 ρ 是从 Ctr_2 到 Ctr_1 向上的仿真:

(1) ρ 保持初始条件: $\exists n_1,n_2\cdot(Init_2;(g_T^2\ \&\ D_T^2)^{n_2};\rho)\Rightarrow Init_1;(g_T^1\ \&\ D_T^1)^{n_1}\wedge\ 0\le n_1,n_2<\infty$.

(2) 对接口中所有的方法 m, ρ 保持卫式条件的等价: $[\exists u\cdot\rho(v,u)\wedge\forall m\in I_1.MDec\cdot(g_m^1(u)\equiv g_m^2(v))]$.

(3) 对于接口中每种方法 m 对应的设计有:

$$\exists n_{1m},n_{2m}\cdot(g_m^2\wedge D_m^2);(g_T^2\wedge D_T^2)^{n_{2m}};sim\sqsupseteq sim;(g_m^1\wedge D_m^1);(g_T^1\wedge D_T^1)^{n_{1m}}\wedge 0\le n_{1m},n_{2m}<\infty,$$

其中, $sim\overset{\text{def}}{=}\text{true}\vdash(\rho(v,u')\wedge wait'=wait)$.

契约之间的精化关系是关于契约的全局性质.上述仿真技术证明精化关系的定理,把对契约全局性质的证明转化到对契约一组局部性质的证明,即对两个契约中对应方法功能规约的比较,从而简化了证明过程.

定理 5(仿真完备性). 如果 $Ctr_1\sqsubseteq_{ctr}Ctr_2$,那么存在一个契约 Ctr,满足:

(1) 存在一个向上的仿真从 Ctr 到 Ctr_1;

(2) 存在一个向下的仿真从 Ctr 到 Ctr_2.

本节给出了契约的两个模型,基于状态的模型和基于事件的模型,并给出了保证它们之间一致性的方法.可以分别运用两种模型对接口的功能和交互两个特征进行建模和分析,实现了关注点分离.同时,这样的方式也赋予用户极大的灵活性,从技术角度来看,用户可以选取他们最熟悉的模型、方法和工具来研究构件的行为.从工程实践来看,不同类型的建模方法适用于解决不同的问题,构件开发过程通常选取基于状态的模型来指导他们的开发活动,构件组装过程则运用基于事件的模型来保证不会发生死锁和发散.

2.2 构件的形式化模型和精化方法

构件是对其服务接口契约的一种实现,它包含了实现该规约所必需的代码.构件的实现通常要求环境提供必要的支持,实现代码会以方法调用的方式获得环境提供的服务.构件对外部方法的依赖定义了构件的需求接口.

定义 13(构件). 一个构件是一个多元组 $(I,Init,Tau,TCode,MCode,InMDec)$,其中:

(1) I 是一个接口.

(2) $Init$ 是一个设计,用来定义构件的初始化状态,形如 $\text{true}\vdash init(I.FDec')\wedge\neg wait',init$ 是一个谓词.

(3) $TCode$ 是一个集合,它包含一组构件自主行为的代码实现.构件自主随机选择并执行 $TCode$ 中的一个元素.

(4) Tau 是一个卫式设计 $(I.FDec\cup I.FDec',g_T\ \&\ D_T)$,它定义了构件内部的自主行为,即当卫式条件 g_T 为真时,构件自动地执行设计 D_T 定义的行为.可以从自主行为的代码计算它对应的卫式设计,用 $\sqcap$ 连接 $TCode$ 每个元素对应的卫式设计.

(5) 函数 $MCode$ 映射构件的服务方法到它们的代码实现.

(6) $InMDec$ 是被构件的实现调用的外部方法的声明,它体现了构件对外部环境的依赖.

为了便于使用,在下文中用 $C.I,C.Init,C.Tau,C.MCode$ 和 $C.InMDec$ 来引用构件 C 中对应的部分.

构件的语义描述其服务接口与需求接口之间的语义依赖关系,并研究如何根据代码实现,从其需求接口的契约计算其服务接口的契约.

定义 14(构件的语义). 构件 C 的语义定义为一个函数,对于每一个满足条件 $InCtr.MDec=C.InMDec$ 的契约 $InCtr$, $C(InCtr)$ 是一个契约 $OutCtr$,这里, $\sqcap$ 表示非确定性选择.

$$OutCtr.FDec \overset{\text{def}}{=} C.FDec \cup InCtr.FDec,$$
$$OutCtr.MDec \overset{\text{def}}{=} C.MDec \cup InCtr.MDec,$$
$$OutCtr.Init \overset{\text{def}}{=} C.Init \wedge InCtr.Init,$$
$$OutCtr.Tau \overset{\text{def}}{=} C.Tau \sqcap InCtr.Tau,$$
$$OutCtr.MSpec \overset{\text{def}}{=} \Phi.$$

函数 Φ 为集合 $C.MDec \cup InCtr.MDec$ 中的每种方法 m 指定一个卫式设计:

(1) $\Phi(m) \overset{\text{def}}{=} InCtr.MSpec(m)$,如果 $m \in C.InMDec$.

(2) $\Phi(m) \overset{\text{def}}{=} g_m \,\&\, [\![body(m)]\!]$,如果 $m \in C.MDec$. 其中,$[\![body(m)]\!]$ 是从 m 实现的命令序列计算出来的反应式设计.

定理 6. 一个构件 C 是从其需求接口契约到服务接口契约的单调上升函数.若 $Ctr_1 \sqsubseteq_{ctr} Ctr_2$, 则 $C(Ctr_1) \sqsubseteq_{ctr} C(Ctr_2)$.

对于构件,总是期望它对配置环境的要求越弱越好,同时提供的服务越强越好.通过比较对于相同的输入契约,不同构件输出的服务契约之间的精化关系,可以定义构件之间的精化关系.

可以利用精化关系研究构件与环境之间的相容性.设环境提供的服务接口的契约为 Ctr_s, 那么将构件 C 插入环境之后,可计算构件 C 服务接口的契约是 $C(Ctr_s)$. 如果 $C(Ctr_s) \sqsupseteq Ctr_r$, 即构件 C 服务接口的契约是环境要求构件提供的契约 Ctr_r 的精化,那么,可以认定,构件和环境是相容的.这里,环境可以是一个构件,也可以是一组构件组合而成的复合构件.

定义 15(构件的精化). 一个构件 C_1 被构件 C_2 精化,表示为 $C_1 \sqsubseteq_{comp} C_2$, 如果它们满足如下条件:

(1) 它们有相同的服务接口: $C_1.MDec = C_2.MDec$.

(2) 它们有相同的需求接口: $C_1.InMDec = C_2.InMDec$.

(3) 对于所有的输入契约 $InCtr, C_1(InCtr) \sqsubseteq_{ctr} C_2(InCtr)$ 都成立.

2.3 构件的组装

构件的组装过程是两个构件各自在对方的服务接口中寻找可与自己的需求相匹配的方法,并将它们连接起来.在运行时刻,构件双方需要在相连方法的调用和返回事件上同步.即当函数调用发生时,发出调用方的调用事件需要与接受调用方的接受事件同步;函数返回时,接受调用方的返回事件需与发出调用方的接受事件同步.

定义 16(构件的组装). 设构件 C_1 和 C_2 的域变量声明互不相交,即 $C_1.FDec \cap C_2.FDec = \varnothing$, C_1 和 C_2 组装后构件 $C_1 \oplus C_2$ 定义为

$$(C_1 \oplus C_2).FDec \overset{\text{def}}{=} C_1.FDec \cup C_2.FDec,$$
$$(C_1 \oplus C_2).MDec \overset{\text{def}}{=} (C_1.MDec \cup C_2.MDec) \setminus (C_1.InMDec \cup C_2.InMDec),$$
$$(C_1 \oplus C_2).Init \overset{\text{def}}{=} C_1.Init \wedge C_2.Init,$$
$$(C_1 \oplus C_2).Tau \overset{\text{def}}{=} C_1.Tau \sqcap C_2.Tau,$$
$$(C_1 \oplus C_2).TCode \overset{\text{def}}{=} C_1.TCode \cup C_2.TCode,$$
$$(C_1 \oplus C_2).MCode \overset{\text{def}}{=} C_1.MCode \cup C_2.MCode,$$
$$(C_1 \oplus C_2).InMDec \overset{\text{def}}{=} (C_1.InMDec \setminus C_2.MDec) \cup (C_2.InMDec \setminus C_1.MDec).$$

其中,$X \setminus Y$ 表示从集合 X 删除所有在集合 Y 中出现的元素.

例 4:考虑以下两个构件的组合:

构件 *Cal* 的定义如下:

$$FDec=\varnothing,$$
$$MDec=\{square(in\ \ x:INT,out\ \ y:INT)\},$$
$$InMDec=\varnothing,$$
$$Init=skip,$$
$$Tau=\text{false}\ \&\ skip,$$
$$MSpec(square)=\text{true}\ \&\ \text{true}\vdash y'=x*x.$$

构件 *Broker* 声明了这样的服务接口:

$$REQUEST=\{id:INT,param:INT,result:INT\},$$
$$I.FDec=\{requests:[REQUEST],results:[REQUEST]\},$$
$$I.MDec=\{put(in\ x:INT,out\ \ y:INT),get(in\ id:INT,out\ y:INT)\}.$$

Broker 需求接口声明了如下函数:

$$\{square(in\ \ x:INT,out\ \ y:INT)\}.$$

Broker 服务接口的契约定义如下:

$$Init=requests'=\varnothing\wedge results'=\varnothing,$$
$$Tau=requests\neq\varnothing\ \&$$
$$\text{true}\vdash\exists req:REQUEST\cdot req\in requests\wedge[\![square(req.param,req.result')]\!]\wedge$$
$$req.id'=req.id\wedge requests'=requests\setminus\{req\}\wedge results'=results\cup\{req\},$$
$$MSpec(put)=|requests|+|results|<10\ \&$$
$$\text{true}\vdash\exists req:REQUEST\cdot req.id'\notin idset\wedge req.param=x\wedge y'=req.id'\wedge$$
$$requests'=request\cup\{req\}\wedge results'=results,$$
$$MSpec(get)=\exists req:REQUEST\cdot req.id=id\wedge req\in results\ \&$$
$$\text{true}\vdash\exists req:REQUEST\cdot req.id=id\wedge req\in results\wedge y'=req.result\wedge$$
$$requests'=request\wedge results'=results\setminus\{req\}.$$

这里, $idset=\{req.id:INT\mid req\in requests\vee req\in results\}$.

活动 *Tau* 中出现的 $[\![square(req.param,req.result')]\!]$ 是函数调用语句 $square(req.param,req.result')$ 对应的设计,它定义为 $\exists x,y\cdot skip;x'=req.param;g\ \&\ D;req.result'=y;\exists x',y'\cdot skip$ [4],其中, $g\ \&\ D$ 是方法 $square(x,y)$ 的功能规约.

根据定义 16 可以得到构件 $Cal\oplus Broker$, 依据定义 14 可以计算其服务接口的契约,根据定理 3 不难证明, $Cal\oplus Broker$ 是例 2 中可以缓冲 10 个请求的构件的一个精化.这里,构件 *Cal* 负责完成计算任务,构件 *Broker* 负责缓冲请求和结果.在分布式系统中,为了适应网络环境,往往要求构件有一定的缓冲能力,在本例中,这样的构件是通过组合缓冲构件和计算构件来实现的.

本例也说明了如何运用构件的精化定理和构件的组合规则支持分布式构件系统的开发.分布式构件系统的开发是一个自底向上的过程,不断地由一组较小的构件组合成一个较大的构件.在这样的过程中,可以先应用构件的组合规则得到复合构件,并计算它的语义,然后再判定复合构件与设计的规约之间的精化关系.不断重复这样的过程,并在上一层的验证工作中直接使用复合构件的语义,直到验证最终组合而成的系统是系统规约的一个精化为止.

3 相关工作

在分布式环境中,主动构件形式化模型的研究是近年来构件技术研究的重要方向.文献[6]把构件定义为一个关于流的处理函数,构件把输入的一组动作流映射到一组输出动作流,并以流函数间的关系研究构件之间的组合与精化关系.流处理函数模型和本文关于构件动态行为的发散、失败模型都是利用事件模型研究构件.事件模型适于处理构件之间的交互,但事件模型仅仅描述了构件的交互行为,无法由这样的规约直接导出构件的

实现.文献[7]用接口自动机定义接口的语义,用消息序列图定义场景规约,利用接口自动机的组合研究构件的组合,并给出了检验构件行为是否与场景规约相一致的方法.文献[8]用转换系统为构件建模,构件的连接器被定义为对构件之间交互方式设定一种约束.在上述两个工作中,构件服务接口上的事件序列和需求接口上的事件序列被混合在一起进行处理,本文的方法采取服务接口和需求接口分开处理的策略,可以更好地支持基于构件的开发过程中自底向上的构造方式.Reo[9]用抽象数据类型统一处理构件和连接器的语义.抽象数据类型非常适于定义构件的规约,但是文献[9]没有给出如何利用抽象数据类型定义程序命令的语义.本文利用设计作为构件形式化的基础,可以方便地在设计这一数学系统中证明构件规约与构件实现之间的精化关系,直接支持构件的开发活动.文献[10]用 B 机器定义构件的语义,用 B 机器的精化方法研究构件之间的精化关系,这可以类比本文关于构件契约的定义和通过仿真证明精化关系的定理.文献[11]应用基于事件的异步交错执行模型研究了构件的语义、构件的组合以及对复合构件的验证.构件的语义包含了构件对应的执行表示(通常是进程代数中的一个进程)和以(需求性质,确保性质)这种方式出现的对构件性质的描述.构件的性质可以和构件一起被组装.文献[11]中考虑的构件组装方式可以用本文的构件组装方法直接加以处理.

本文借鉴了文献[4,12]中利用构件演算为反应式构件构造形式化模型的方法.文献[4]中描述的反应式构件是一种以被动方式工作的构件,它没有自主行为,完全被动地对外界环境的调用作出反应.反应式构件是在一个较高的抽象层次上为构件建模,它对构件复杂的交互行为进行了一定的抽象,适于在构件系统开发的早期阶段分析构件系统的性质,指导构件系统的设计.文献[12]展示了这样的一个设计过程.它研究了在面向服务的构架中,利用反应式构件为服务建模,并用反应式构件的精化方法支持对较大的服务进行功能分解,得到一组较小服务的过程.本文研究的主动式构件是在更加靠近构件实现的层次上对构件进行建模和分析.在实际的分布式系统中,具有自主行为能力的主动构件更具有普遍性.在这个层次上研究构件,就需要为构件的自主活动建模,需要分析和处理自主活动带来的构件整体行为上的非确定性.与本文相同,文献[6,7,9,11]都明确地将构件作为一种主动的软件实体来进行建模和分析,并研究了主动构件行为的非确定性问题.文献[13]将构件划分成被动的反应式构件和主动的进程构件两种类型,这里,进程构件不提供服务方法,因而不能被调用,只能调用其他反应式构件的服务方法.由于不能直接处理带有服务接口的主动构件,文献[13]对两种类型构件组装的顺序进行了限制.可以用本文主动构件的形式模型统一处理这两种类型的构件,并对按照任意顺序组装而成的复合构件进行分析.同时,本文中有关精化关系的定理可以支持反应式构件和主动构件之间精化关系的判定.反应式构件是规约,一组主动构件组合而成的复合构件是对该规约的一个精化.这种精化方法可以运用在把概念层次上的构件在实际的构件模型上加以实现的过程中,保证产生的构件产品的正确性.

4 总结及展望

在基于构件的软件开发过程中引入精化方法,从而保证构件系统的正确性是构件技术研究的重要课题.本文研究了分布式环境中主动构件的形式模型和精化方法.以构件演算为基础,在契约中加入用卫式设计描述的构件自主活动的功能规约,并通过重新定义契约的协议和契约发散、失败集合对的计算方法,使得自主活动对构件的影响可以直接反映在主动构件的交互行为之中.契约的精化关系定义在契约发散、失败集合对的包含关系之上,本文给出并证明了用仿真技术证明精化关系的定理.通过将构件的语义定义为从其需求接口的契约到其服务接口的契约,可以利用契约的精化来证明构件精化.研究了组装规则实现构件的并发式组装,它与构件精化方法的综合应用可以实现整个系统由构造保证正确.

今后的工作包括研究如何设计新的工具以及应用已有的工具支持本文中的演算方法.向此形式模型中引入时间模型,处理构件与时间相关的特性,也是下一步工作的重要内容,应用此方法对主流的构件系统,如基于 CORBA 的构件系统做实例研究也在进行之中.

致谢 本文的主要工作是作者作为 Fellow 在 UNU/IIST 访问学习期间完成的,文中很多的建模思想和技术方法

都来源于与何积丰高级研究员和刘志明研究员的讨论,华东师范大学软件学院的刘静教授对本文的初稿提出了很多有益的建议,在此对他们表示衷心的感谢.

References:

[1] Morgan C. Programming form Specifications. Hemel Hempstead: Prentice-Hall, Inc., 1998. 163−179.

[2] He JF, Li XS, Liu ZM. rCOS: A refinement calculus of object systems. Theoretical Computer Science, 2006,365(2):109−142.

[3] Hoare CAR, He JF. Unifying Theories of Programming. London: Prentice-Hall, Inc., 1998. 74−85.

[4] He JF, Li XS, Liu ZM. A theory of reactive components. Electronic Notes in Theoretical Computer Science, 2006,160:173−195.

[5] Roscoe AW. Theory and Practice of Concurrency. London: Prentice-Hall, Inc., 1997. 519−553.

[6] Bergner K, Rausch A, Sihling M, Vilbig A, Broy M. Foundations of Component-Based Systems. New York: Cambridge University Press. 2000. 189−210.

[7] Hu J, Yu XF, Zhang Y, Zhang T, Wang LZ, Li XD, Zheng GL. Scenario-Based specifications verification for component-based embedded software designs. In: Proc. of the ICPP 2005 Workshops. IEEE Computer Society, 2005. 240−247.

[8] Gössler G, Sifakis J. Composition for component-based modeling. In: de Boer FS, Bonsangue MM, Graf S, de Roever WP, eds. Proc. of the FMCO 2002. LNCS 2852, Heidelberg: Springer-Verlag, 2003. 443−466.

[9] Arbab F. Abstract behavior types: A foundation model for components and their composition. In: de Boer FS, Bonsangue MM, Graf S, de Roever WP, eds. Proc. of the FMCO 2002. LNCS 2852, Heidelberg: Springer-Verlag, 2003. 33−70.

[10] Chouali S, Heisel M, Souquières. Proving component interoperability with B refinement. Electronic Notes in Theoretical Computer Science, 2006,160:157−172.

[11] Xie F, Browne JC. Verified systems by composition from verified components. In: Proc. of the ESEC/SIGSOFT FSE 2003. New York: ACM, 2003. 277−286.

[12] Liu J, He JF, Liu ZM. A strategy for services realization in service-oriented design. Science in China (Series F), 2006,49(6): 864−884.

[13] Chen X, He JF, Liu ZM, Zhan NJ. A model of component-based programming. In: Arbab F, Sirjani M, eds. Proc. of the FSEN 2007. LNCS 4767, Heidelberg: Springer-Verlag, 2007. 191−206.

陈鑫(1975－),男,辽宁大连人,博士生,主要研究领域为软件工程,面向对象的技术,构件技术,形式化方法.