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.