计算机科学 1999Vol 26№. 12

软件体系结构的一种组合模型*)

A Composition Model for Software Architectures

焦文品 史忠植

(中国科学院计算技术研究所 北京 100080)

Abstract In this paper, a formal composition model for software architectures is put forward. In the model, components and connectors are described as independent processes, and constraints which components and connectors should satisfy while composing are expressed in temporal logic formula.

Keywords Software architecture, Composition model, π-calculus, Temporal logic

1 前 言

当前的各种应用系统,无论是其拓扑结构,还是其运行平台,以及其随后的进化过程,都具有开放性。一个开放系统可以用如下公式表示[3]:

OpenSystem=Components+Coordination

其中,Component 为构成软件系统的组件,是具体且实在的组成成分;而 Coordination 则是用来将各种组件联结在一起的黏合剂,即通常所说的连接方式(或连接器)。

软件体系结构明确刻画了特定系统是如何由其组件构成的[4],其中每个组件除了具有独自的行为规范和特征外,相互之间还存在特定的协作规范。

如何由较小的子系统构筑整个软件系统,即软件组合已成为软件体系结构研究中亟待解决的关键问题之一[6]。软件组合刻画了构成系统的组件的界面,以及组件间的组合机制和组合规则。但目前的研究中,不仅组件的类型贫乏(主要是基于对象的),组件界面的定义难以统一,而且组件间的组合机制也显得单调乏力,组合规则更缺乏明确的描述。如在以 π 演算[12]为形式化基础的方法中[3,7],组件被刻画成进程,组件间通过 π 演算提供的组合操作来实现组合。在文[9]中,组件是用 TLA[6]刻画的独立进程,组件间通过共享内存变量建立联系,并通过进程组合将相互依存的进程联合起来,另外还有一类将组件看成一个模块,并通过向外界提供或请求服务来与外界发生关系[8,10]。

我们发现利用谓词逻辑和时序逻辑来刻画组件时(如文[9]),组件的内涵比较丰富,但组合方式很单一,而利用进程代数来刻画组件时(如文[7]),组件间的组合比较灵活,但组件的内涵匮乏,很难刻画组件的不变式性质以及各种非功能性的性质,如性能、实时性、安全性等。

因此,在本文中,我们试图从软件体系结构的角度来研究软件的组合。在形式化组件的同时,将组件间的组合方式即连接方式以独立的实体给予形式化。另一方面,在形式化过程中,我们在两个层次上着手:在第一层面上,用多价 π 演算(见附录)来刻画体系结构元素(组件及连接方式)间的组合,在次一层面上,用时序逻辑来描述元素的内部特性及其得以施用的环境规范。

2 体系结构构成元素

在 P&W[11]体系结构模型中,构成体系结构的元素有处理元素、数据元素及连接元素三种。其中,处理元素是用来处理数据元素的,而数据元素则为处理元素提供数据信息来源,我们将这两者统称为体系结构的组件;而将连接元素称为连接方式或连接器。

2.1 组件

一般认为,组件就是提供某种功能的计算单元。但事实上,有些组件主要用来保存系统的状态信息,或为其它功能性组件提供计算得以进行的场所,如数据结构、数据文件、共享资源等。另一方面,从软件体系结构的角度来看,组件要与其它组件进行组合或合作,除了具有计算功能,能满足系统的功能性要求外,还必须具备相互适应的结构性特征,以及能满足系统的各种非功能性要求的性质,如实时性、安全性等[2]。

组件可能具有某种类型或子类型[1],但组件与对象又有所不同[5]。组件是特地为软件组合而设计的,必

*)由 863 高科技项目资助。

须具备定义明确的组合界面，如转入/转出(Import Export)服务等。另外，组件可以是原始的，也可以是组合的[1]。鉴于组合组件是通过下文将要描述的组合机制和组合规则组合而来的，并且组合后的最终结果仍可以被看作成一个原始组件参与新一轮的组合，所以在形式化定义组件时，我们不区别原始或组合组件。

定义 1 组件(Component)为一个六元组：

$$Component =_{def} (Interface, Players, GC, EC, SubComp, Spec)$$

其中，Interface 为组件向外界提供的服务；Players 是用来与连接器进行连接的端口。因为组件要与其它组件发生联系总是通过连接器进行的，所以我们没有把它需要外界所提供的服务列入 Interface 中，而是放在 Players 中，这样 Players 由两部分组成：插向连接器的负端口(OutP，相当于模块的转出功能)以及供连接器插入的正端口(InP，相当于模块的转入功能)。且 $Interface =_{def} \{Export\text{-}Services\}$；$Players = OutP + InP$；$OutP \cap InP = \varnothing$；$Players \cap Interface = \varnothing$。

GC 定义了组件对 OutP 的限制，即规定在什么(内部)条件下 OutP 能嵌入连接器的 Role 中。它用时序逻辑公式来刻画：$GC =_{def} GC : OutP \rightarrow TLF(OutP)$。其中，TLF(OutP)表示定义在 OutP 上的时序逻辑公式，即该公式中的自由变元 $fv(TLF(OutP)) = OutP$，则 GC(p)，p：OutP，为组件对负端口 p 的限制条件。

EC 则定义了组件对 InP 的要求，即规定在什么(外部)环境下可以让连接器的 Role 嵌入对应的 InP 中。它也用时序逻辑公式来刻画：$EC =_{def} EC : InP \rightarrow TLF(InP)$。

SubComp 指出组合组件所包含的子组件是什么。若组件为原始组件，则此集合为∅。

Spec 刻画了组件界面 Interface 的实现规范，以及与组件相关的非功能性性质。它们都用时序逻辑公式来刻画：$Spec =_{def} \{TLF(i) \mid i : Interface\} \cup \{nonFunctionalPreperties\}$。

2.2 连接方式(连接器)

人们在研究软件组合时，往往侧重于组件，而对于组件间的交互，即连接方式，由于它们一般隐含或分布于系统之中，因而缺乏明确的描述[1]。

连接方式用来刻画组件间的联系，即组件间的交互协议，并为这些组件提供组合手段[2]。与具体而实在的组件不同，连接器可能并不对应于某个独立的编译单元。

但人们在研究软件体系结构时发现，把连接器从组件中独立开来予以单独考虑会带来很多好处[3,4]，如可提高组件的可重用性、易维护性，简化对体系结构风格或模式的理解，为分析系统的整体行为提供更好的手段等。

定义 2 连接器(Connector)为一个四元组：

$$Connector =_{def} (Roles, Protocol, IC, OC)$$

其中，Roles 是用来与组件的 Players 连接的端口，与 Players 相似，连接器中的 Roles 也分为正负端口(InR，OutR)，其中 InR 将与组件中的 OutP 互连，而 OutR 则与 InP 互连。且：$Roles = InR + OutR$；$InR \cap OutR = \varnothing$。

IC 和 OC 分别与组件中的 GC 和 EC 相对应，用来规定在什么条件下允许组件的 OutP 插入 InR 中，或 OutR 插向组件的 InP 中。它们也都用时序逻辑公式描述：$IC =_{def} IC : InR \rightarrow TLF(InR)$；$OC =_{def} OC : OutR \rightarrow TLF(OutR)$。

Protocol 与组件中对界面的规范 Spec 相似，但它刻画的是连接器中 Roles 应遵循的协议，如端口之间的内部联系，以及连接方式的转换等。$Protocol =_{def} \{TLF(roles) \mid roles \in Roles\}$。

组件与连接器间的关系可以用图 1 来示意，图中组件的 Interface 一部分构成组合组件的 Interface，而另一部分则映射成组合组件的 Players。

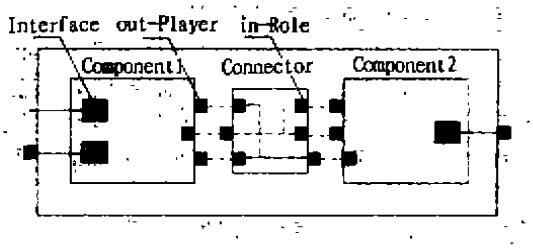

图 1 组件与连接器关系示意图

3 开放系统的组合

在前面提到，组件和连接器可被刻画成进程来参与体系结构的组合，一旦将组件和连接器都定义成进程后，进程间可通过链接($\overline{Link}$)来传输组件或连接器的端口，这样可以将各种连接方式以并发进程间的组合这一形式统一起来。下文我们将用多价 π 演算[2]来定义组件和连接器进程。

3.1 组件进程

当组件被刻画成进程时，组件向外界提供的服务(Export)通过进程的负链接($\overline{link}$)输出给其它组件进程，而需要其它组件提供的服务(服务请求，Import)则通过进程的正链接输入。因为组件向外界请求的服务是通过连接器间接得到的，它无须知道谁能提供所需的服务，但一旦通过连接器找到所需的服务后，为了能

正确将获得的服务返回给请求者，服务将用于何处的位置信息也要随请求一起传递。

服务请求子进程可以非形式化地描述为：哪儿请求，请求什么，用多价 π 演算定义为：

$$Req(w,r)=_{def}\bar{r}(w,r')$$

其中 w 为获取服务的位置，r:InP，式子右端的 r 为以正端口命名的链接名，而 r' 为具体的请求，它是带参数的 Import 服务。从下文的各类进程定义中将会看到，此输出链接中输出的第二个名字似乎可以省去，但考虑到服务请求在通过连接器传递给其它组件时，可能需要进行某种形式的变换，如参数变换、当多个请求对应一个服务时将请求参数化等，因此我们在组件进程中保留了该名字。

类似地，提供服务子进程可以非形式化地描述为向谁提供，提供什么。用多价 π 演算定义为：

$$Prov(x,s)=_{def}!\ (vy)s(x,y)\cdot(\bar{x}(s'))$$

其中 x 表示服务将送往何处，y 为请求名，而 s:OutP 为组件提供的对应服务。式子右端的 s 为以负端口命名的链接名，而 s' 为提供的具体服务，它是带参数的 Export 服务。在此子进程中出现 y 一方面是出于多价 π 演算要求参与通信的抽象和具体对象具有相同的目数(arity)的考虑，另一方面也是参数传递和服务变换的需要，但因为一旦提供服务的子进程收到该请求，并将 y 所带来的信息加入到向外界提供的服务 s' 中之后，其作用将消失，因此我们将 y 定义成一个约束名字。

这样，提供服务 $(OutP_1,\cdots,OutP_m)$ 并请求服务 $(InP_1,\cdots,InP_n)$ 的组件 Comp 可以用多价 π 演算定义为：

$$\begin{aligned}Comp=&Prov(x,OutP_2)\mid\cdots\mid Prov(x,OutP_m)\mid\\&Req(w_1,InP_1)\mid\cdots\mid Req(w_n,InP_n)\mid\\&w_1(InP'_1)\cdot\cdots\cdot w_n(InP'_n)\cdot CompImp\end{aligned}$$

其中 CompImp 为组件的实现，它在 w_i 接收到所请求的具体服务后将请求代以对应的服务。

3.2 连接器进程

在定义连接器时，我们曾指出连接器规定了各 Roles 应遵循的协议，即 Roles 之间的内部联系、以及不同 Role 的相互转接(如某个 OutR 的功能由一个或多个 InR 联合提供；某个 InR 能同时为一个或多个 OutR 提供服务源)等。

设某组件向外界请求的服务 s 是通过连接器上的端口 $OutR_i$ 传递来的，另一方面，该服务 s 又是其它组件通过连接器上的端口 $InR_1,\cdots,InR_n$ 传入并转接到 $OutR_i$ 上的，则此行为可以用一个多价 π 演算进程定义为：

$$OutR_i(x,r)\cdot\overline{InR_1}[x,r']\cdot\cdots\cdot\overline{InR_n}[x,r'']$$

其中 r 为组件所请求的服务，而 r'、r'' 为连接器对 r 的某种变换。

这样，整个连接器进程可以定义为：

$$\begin{aligned}Conn=(vr)&OutR_1(x,r)\cdot\overline{InR_1}[x,r']\cdot\cdots\cdot\overline{InR_1}[x,r'']\mid\cdots\mid OutR_m(x,r)\cdot\overline{InR_1}[x,r']\cdot\cdots\cdot\overline{InR_1}[x,r'']\end{aligned}$$

3.3 装配进程

尽管连接器是用来连接不同组件的，但为了便于组件及连接器的重用，前文所定义的组件和连接器都是相互独立的，因此还需定义组件与连接器端口间的对应关系，这种映射关系用来将组件通过连接器装配起来，同样，我们也用一个多价 π 演算进程来定义这种关系，即装配进程。

设在组件与连接器以及连接器与组件间存在两个映射 α，β，用以表明组件中的每个 InP 或连接器中的每个 InR 都存在对应的 OutR 或 OutP 与之对应。

$$\alpha:\bigcup_{i=1}^{n}Comp_i\cdot InP\rightarrow\bigcup_{i=1}^{m}Conn_i\cdot OutR$$

$$\beta:\bigcup_{i=1}^{m}Conn_i\cdot InR\rightarrow\bigcup_{i=1}^{n}Comp_i\cdot OutP$$

这样，InP 与 $OutR=\alpha(InP)$ 间的关系可以表示为子进程：

$$InP(x,y)\cdot\overline{OutR}[x,y]$$

类似地，InR 与 $OutP=\beta(InR)$ 间的关系可以表示为子进程：

$$InR(x,y)\cdot\overline{OutP}[x,y]$$

于是用来规定所有 $InP_i(i=1\cdots N)$ 与 $InR_j(j=1\cdots M)$ 的映射关系的整个装配进程定义为：

$$\begin{aligned}Attachment=&\prod_{i=1}^{N}(InP_i(x,y)\cdot\overline{\alpha(InP_i)}[x,y])\mid\\&\prod_{i=1}^{M}(InR_i(x,y)\cdot\overline{\beta(InR_i)}[x,y])\end{aligned}$$

3.4 系统的组合

在定义好构成系统所需的组件、组件间的连接方式以及组件与连接器的装配关系之后，就可以用它们来组合产生整个系统。

设系统由组件 $(Comp_1,\cdots,Comp_n)$ 通过连接器 $(Conn_1,\cdots,Conn_m)$ 组合而成，参与组合的组件和连接器端口间的映射关系为 α，β，它们对应的装配进程为 Attachment，那么系统的组合可以定义为：

$$\begin{aligned}\textbf{Composition}=_{def}&Comp_1\mid\cdots\mid Comp_n\mid Conn_1\mid\\&\cdots\mid Conn_m\mid Attachment\end{aligned}$$

但这种组合仅是形式上的。在组合过程中，组件与连接器之间是否会出现冲突，即对应的插口能否允许由 α 和 β 所映射的端口正确地插入，即能否正确使用外界提供的服务呢？

根据 Assumption/Guarantee(环境假设/内部保证)特性[14]，可以断言——且某个组件的环境假设 E 得

到满足，它必然满足其得以正确运行的内部保证条件 $G^{[-0]}$。

定义 3 设系统中存在组件 Comp，若 $p \in InP$. Conn. $OC(\alpha(p)) \Rightarrow$ Comp. $EC(p)\{\alpha(InP)/InP\}$，则说组件 Comp 端口 p 允许连接器中的插头 $\alpha(p)$ 正确插入；同样，对于系统中的连接器 Conn，若 $r \in InR$. Comp. $GC(\beta(r)) \Rightarrow$ Conn. $IC(r)\{\beta(InR)/InR\}$，则说连接器 Conn 端口 r 允许组件中的插头 $\beta(r)$ 正确插入。其中 $F\{y/x\}$ 表示将公式 F 中集合 x 内所有元素的自由出现都替换成集合 y 内对应的元素。

组合命题：设完备的组合系统 S 由组件 $Comp_1, \cdots, Comp_n$ 通过连接器 $Conn_1, \cdots, Conn_m$ 互连而成，参与组合的组件和连接器端口间的映射关系为 α, β，若：(1) $\forall i:1\ n \forall p \in InP_i \exists j:1\ m (\alpha(p) \in Conn_j.OutR \Rightarrow (Conn_j.OC(\alpha(p)) \Rightarrow Comp_i.EC(p)\{\alpha(InP)/InP\})$，且 (2) $\forall j:1\ m \forall r \in InR_j \exists i:1.\ n (\beta(r) \in Comp_i.OutP \Rightarrow (Comp_i.GC(\beta(r)) \Rightarrow Conn_j.IC(r)\{\beta(InR)/InR\})$。则 $Comp_1, \cdots, Comp_n$ 可以通过连接器 $Conn_1, \cdots, Conn_m$ 协调地组合成系统 S。

证明：对于完备系统，即系统内部的各组件和连接器的正端口都存在对应的负端口与之互连。根据定义 3，命题中第(1)个蕴含式说明组件能够正确地使用连接器所提供的服务，而第(2)个蕴含式则说明连接器也能正确地使用组件所提供的服务。由此可见，在组合系统 S 中各组件和连接器都能正常运行。 □

3.5 组合系统

上一部分讨论了系统的组合方式，下面将进一步研究组合后的系统所具有的性质，以及新的组件如何才能成为参与新一轮的组合。

在定义组件时，我们已经指出，组件的界面 Interface 一部分映射成组合组件的 OutP，而另一部分则成为组合组件的 Interface。

设系统 CS 由组件 $Comp_1$ 和 $Comp_2$ 组合而成，则有：

CS. SubComp $= \{Comp_1, Comp_2\}$
CS. Interface $\subseteq Comp_1.Interface \cup Comp_2.Interface$
CS. OutP $\subseteq Comp_1.Interface \cup Comp_2.Interface$，且
CS. Interface + CS. OutP = $Comp_1.Interface \cup Comp_2.Interface$

因为组件的规范 Spec 是时序逻辑公式，它可以直接作为组合组件的 Players 的限制条件，所以有：

CS. Spec $\Leftarrow Comp_1.Spec \cup Comp_2.Spec$
CS. GC $\Leftarrow Comp_1.Spec \cup Comp_2.Spec$

一般说来，一个系统要么是封闭的，即不受外界环境的制约；要么是开放的，即系统必须适应它所处的环境，并只有在特定环境下才能正常运转。若 CS 为封闭系统，则 CS. Players $= \varnothing$；若 CS 为开放系统，则 CS. OutP + CS. InP $\neq \varnothing$，但其 InP 及 EC 无法从子组件中获得，需要重新定义。

上述结论只有当组合系统的界面 Interface 及其所提供的组合端口直接来自其组件时才适用。当组合系统的界面或端口需要对组件 Interface 中的原始对象进行某种形式的重组时，组合系统的界面或端口就需要重新定义。

设在图 2 所示的组合系统中，I_{01} 为 (I_{11}, I_{21})，而 P_{01} 为 (if B then I_{12} else I_{22}; I_{23})。则 CS. Spec $|I_{01} \Leftarrow Comp_1.Spec|I_{11} \wedge Comp_2.Spec|I_{21}$，而 CS. GC $(P_{01}) \Leftarrow (B \wedge Comp_1.Spec|I_{12} \vee \sim B \wedge Comp_2.Spec|I_{22}) \wedge Comp_2.Spec|I_{23}$。式中 $Spec|I$ 表示与 I 有关的规范。

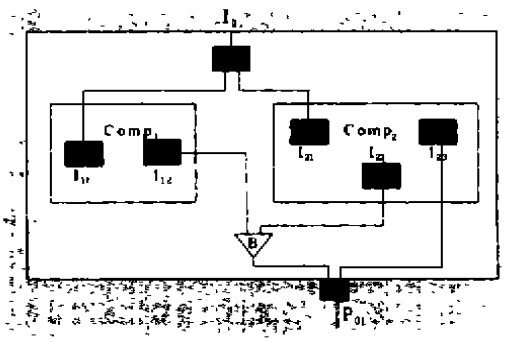

图 2 开放系统对其子组件界面所提供的原始对象重新组合示意图

结束语 在本文所描述的组合模型中，组件和连接器都被看成独立的实体，它们参与组合时所需满足的内外部条件由那些对端口的限制条件来体现。从定义中可以看出，用来定义对端口的限制的时序逻辑公式中只涉及组件和连接器间相互能看到的 Players 和 Roles。这较好地保证了它们的独立性，通过将组件及连接方式刻画为进程，可以将各种不同的组合方式转化为统一的形式，从而简化了模型的组合机制。而组件和连接器的内部规范是通过时序逻辑公式来刻画的，这样为保证组合系统的一致性和正确性提供了逻辑基础。

在本文的最后，利用所提出的组合模型，简要地说明了如何从组件中推导产生组合系统的整体特性。

在定义组件和连接器时，我们曾指出组件可以是原始的，也可以是组合的。一旦将连接方式看成为独立的实体后，连接器就会象组件一样，既可以具有自己的类型或子类型，也存在原始和组合连接器之分，如过程调用、消息传递、共享数据等都是原始的连接方式，而事件驱动则为组合的。但由于连接器是用来连接组件的一种中间媒介，它不是被连接的对象，显然组合的连

接方式不能通过组合模型中所提供的组合机制和规则在原始连接方式的基础上组合而来。这似乎说明组合连接方式的产生需要新的机制，这将是我们进一步所要研究的问题之一。

在组合模型中，我们将进程代数和逻辑揉和在一起，这样做会带来一定的好处，但我们希望在不久的将来能将该模型建立在统一的形式化基础之上。

参考文献

1 Shaw M, Garlan D. Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, 1996

2 Shaw M. Truth vs Knowledge: The Difference Between What a Component Does and What We Know It Does. In: Proc. 8th Intl. Workshop on Software Specification and Design. March 1996

3 Nierstrasz O, et al. Formalizing Composable Software Systems-A Research Agenda. In: Proc. 1st IFIP Workshop on Formal Methods for Open Object-based Distributed Systems FMOODS'96, Chapmann and Hall, 1996. 271～282

4 Nierstrasz O, Meijler T D. Research Directions in Software Composition. ACM Computing Surveys, 1995, 27(2): 262～264

5 Nierstrasz O, Meijler T D. Requirements for a Composition Language. In: Ciancarini P, et al., Eds, Object-Based Models and Languages for Concurrent Systems, LNCS 924, Springer-Verlag, 1995. 147～161

6 Abd-Allah A, Boehm B. Reasoning about the Composition of Heterogeneous Architectures [USC Technical Report USC-CSE-95-503], 1995

7 Magee J, Kramer J. Dynamic structure in software architectures. In: SIGSOFT'96 Proc. of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering, San Francisco, 1996. 3～14

8 Lamport L. The Temporal Logic of Actions. ACM Toplas, 1994, 16(3): 872～923

9 Abadi M, Lamport L. Conjoining Specifications, ACM Toplas, 1995, 17(1): 507～534

10 Luckham D C, et al. Specification and Analysis of System Architecture Using Rapide. IEEE Trans. on Software Engineering, Special Issue on Software Architecture, 1995, 21(4): 336～355

11 Perry D E, Wolf A L. Foundations for the Study of Software Architecture. ACM SIGSOFT Software Engineering Notes, 1992, 17(4)

12 Milner R. The polyadic Pi calculus: a tutorial. [Technical Report ECS-LFCS-91-180]. UK: Computer Science Department, University of Edinburgh, 1991

13 Lam S S, Shankar A U. A theory of interfaces and modules I-composition theorem. IEEE Trans. on Software Engineering, 1994, 20(1): 55～71

14 Jones C B. Specification and design of (parallel) programs. In: R E. A. Mason, ed. Information Processing 83: Proc. of the IFIP 9th World Congress. North-Holland, Amsterdam, 1983. 321～332

***附录(多价π演算)**

多价π演算是一种很基本的演算，它用通信结构来描述和分析并发系统。在π演算中，系统是由若干个相互独立的通信进程组成的。进程间通过连接一对互补端口的通道或链路来进行通信。在π演算中，名字是最原始的实体，端口、通道或链路都是名字，进程由名字按一定的语法规则组成。多价π演算中的进程有如下几种：

1. 求和$\sum_{i\in I}P_i=P_1+P_2+\cdots+P_n$，它表示选择执行其中的任意一个进程$P_i$，当n=0时，表示结束。

2. 前缀式：$\overrightarrow{yx}.P$，$y(\vec{x}).P$，或$\tau.P$，分别表示在端口输出/输入名字向量x，或先执行一个不可见动作τ，然后再执行P。

3. 组合$P_1|P_2$：即并发地执行进程P_1，P_2，并发是可交换的、可结合的。

4. 限制$(\nu y)P$：它与进程P的行为是相似的，但受到限制的名字y对外界是不可见的。

5. 匹配$[x=y]P$：若名字x与y相同，则执行进程P，否则，结束。

6. 复制$!P$：提供任意个进程P的副本。

7. 结束进程0：它用来表示一个进程的结束。为简单起见，我们在定义进程时，一般将进程最后的0省略掉。

在多价π演算中，计算是通过如下的通信规则来表示的：

$$(\cdots+\overrightarrow{yx}.P)|(\cdots+y(\vec{z}).Q)\rightarrow P|Q\{x/z\}$$

其中，向量x和z必须具有相同的目数。名字向量x通过链路y在进程间传递，通信结束，进程简化为右端形式，同时将Q中所有自由出现的z都用x代替掉。