第41卷 第2期 西 安 交 通 大 学 学 报 Vol.41 №2
2007年2月 JOURNAL OF XI′AN JIAOTONG UNIVERSITY Feb. 2007

动态软件体系结构建模方法研究

于振华[1]，蔡远利[1]，徐海平[2]
(1.西安交通大学电子与信息工程学院，710049，西安；2.马萨诸塞州立大学达特茅斯分校
计算机与信息科学系，02747，北达特茅斯)

摘要：针对用现有方法对动态体系结构建模的不足，提出了一种软件体系结构抽象模型(SAAM). SAAM以两种互为补充的形式化方法——面向对象Petri网OPN和π演算为语义基础，其中OPN可以形象地描述软件体系结构的初始化模型和动态行为，π演算可以描述软件体系结构的动态演化.这两种形式化方法通过SAAM集成在一起，并通过相应的支持工具对模型进行分析.在体系结构演化过程中，SAAM利用π演算的相关分析方法，对组件的演化策略和软件体系结构的一致性进行分析，从而可以在系统开发早期发现体系结构级的错误，提高软件质量.利用SAAM对经典实例自动加油站系统进行的建模与分析表明，SAAM直观、易懂，可以有效地刻画系统动态体系结构.

关键词：软件体系结构；面向对象Petri网；π演算；演化；建模
中图分类号：TP311 **文献标识码**：A **文章编号**：0253-987X(2007)02-0167-05

On Modeling Approach for Dynamic Software Architecture

Yu Zhenhua[1], Cai Yuanli[1], Xu Haiping[2]
(1. School of Electronics and Information Engineering, Xi′an Jiaotong University, Xi′an 710049, China; 2. Department of Computer and Information Science, University of Massachusetts Dartmouth, North Dartmouth 02747, USA)

Abstract: Aiming at the defects of existing methods for modeling of dynamic software architecture, a novel software architecture abstract model (SAAM) is presented, in which two complementary formalisms, namely object-oriented Petri nets (OPN) and π-calculus, are adopted as formal theory bases. The OPN are employed to visualize initial architecture as well as system behaviors; while π-calculus is used to describe software architecture evolutions. These two formal methods are integrated in the SAAM, and the SAAM can be analyzed and verified by the corresponding supporting tools. Furthermore, the evolving strategy of components and the consistency among components can also be analyzed using π-calculus so as to detect the design errors in early software design stage and significantly improve the quality of software. A classical gas station example is used to show that SAAM is intuitional and effectively describes dynamic software architecture.

Keywords: software architecture; object-oriented Petri net; π-calculus evolving; modeling

根据软件体系结构在运行时的演化方式，可以分为静态软件体系结构和动态软件体系结构[1].近年来，已提出一些体系结构描述语言(Architecture Description Language, ADL)和形式化模型[1-4]来描述软件体系结构.只有少数ADL为动态体系结构建模提供了一定的支持，然而它们缺乏对模型相关属性的分析和验证，而这是确保所开发系统具有足够的鲁棒性、一致性、复用性和可维护性的基础.

收稿日期：2006-05-16. 作者简介：于振华(1977～)，男，博士生；蔡远利(联系人)，男，教授，博士生导师. 基金项目：国家高技术研究发展计划资助项目(2003AA721070).

为了给软件体系结构提供一种系统、精确的描述和分析方法，我们扩展了文献[4]的工作，并借鉴了多 Agent 系统体系结构的建模方法，建立了一种软件体系结构抽象模型 SAAM(Software Architecture Abstract Model)来描述系统的静态和动态体系结构. SAAM 以两种互为补充的形式化方法——面向对象 Petri 网(Object-Oriented Petri Nets, OPN)和 π 演算[5]为语义基础，弥补了单独使用 Petri 网和 π 演算的缺陷.

1 软件体系结构抽象模型

由于 OPNADL 不适合描述系统的动态演化，我们引入 π 演算来刻画系统的动态体系结构，并建立软件体系结构的抽象模型 SAAM.

定义 1 SAAM 是一个三元组，$S_M=(C_p, C_n, C_f)$，其中 $C_p=(C_{p,1}, C_{p,2}, \cdots, C_{p,o})$表示软件体系结构中组件的集合，$C_n=(C_{n,1}, C_{n,2}, \cdots, C_{n,p})$表示连接件的集合，$C_f$ 表示体系结构的配置.

组件是一个数据单元或一个计算单元，也是一个三元组，$C_{p,o}=(I_D, O_o, E_p)$，其中：I_D 是组件的标识符集合；O_o 为 OPN 模型中的元，定义了组件的接口和内部实现；E_p 利用 π 演算描述组件的演化.

连接件是组件交互协议的实现，为一个五元组，$C_{n,p}=(L_P, G, K_P, R, E_n)$，其中：$L_P$ 为系统中的智能连接库所，用椭圆表示，负责建立组件之间的消息传递通道，然后组件通过常用的客户/服务器、管道和过程调用等协议进行交互；G 为 OPN 中的元，表示组件间的消息传递通道；K_P 为知识库所，存储系统中组件的相关信息(如名字、地址和接口信息等)；R 是连接件中的角色，为与连接件相交互的、在连接件中处于相同地位的所有组件的抽象集合，$R=\{I_{D,1},\cdots,I_{D,o}\}$，$I_{D,o}$为系统中组件的标志符；$E_n$ 利用 π 演算描述连接件的演化，主要描述连接件建立组件之间交互通道的动态过程.

C_f 为体系结构的配置，主要描述由组件和连接件构成的软件体系结构拓扑，可以促进软件生命周期中不同用户对软件体系结构的理解.

关于组件及连接件的演化机制，将在第 2 节中详细讨论. 软件体系结构模型配置如图 1 所示，主要从宏观层次上描述软件体系结构，侧重于软件系统的整体行为和组件之间的交互，同时也描述软件体系结构模型的静态语义. SAAM 的动态语义可以通过变迁的使能和发射规则来描述.

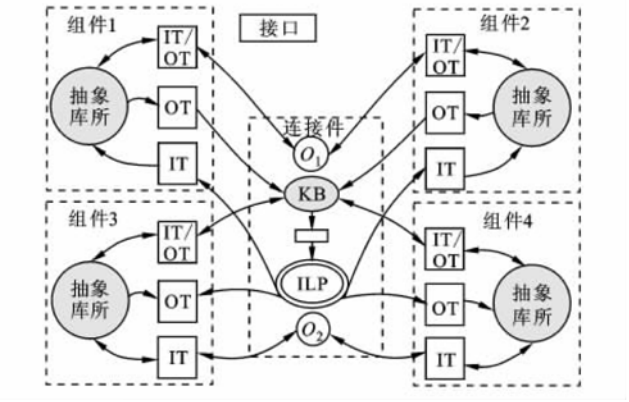

图 1 软件体系结构配置

2 软件体系结构动态演化研究

利用形式化方法建模的一个重要目的就是可以利用现有的数学分析方法对模型进行分析和验证. SAAM 对静态软件体系结构的分析，可以利用 Petri 网的相关分析方法和支持工具(如 INA)对模型进行验证. 本文的重点是分析动态软件体系结构，主要关注组件之间的交互而忽略组件的内部实现细节. 由于组件通过接口进行交互，根据文献[6]，我们使用 π 演算进程表达式来描述组件接口，确保组件的内部实现能完整、正确地由接口来描述，然后在接口层次上分析体系结构的动态演化和一致性.

2.1 软件体系结构动态演化

软件体系结构的动态演化包括组件或连接件的创建或删除、组件的更新、调整负载平衡等几种情况[1]，下面分别讨论.

当新组件加入时，新组件首先把注册信息保存在连接件中，然后通过 I_P 建立和其他组件的交互通道. 组件创建进程为

$$\text{newcomp}(i_d,s)=\text{create}(i_d,s)$$

表示创建了一个标志符为 i_d、服务为 s 的组件. 组件注册进程可以描述为

$$\text{reginfo}(i_d,s)=(i_d,s)(\overline{\text{register}}\langle i_d,s\rangle)$$

表示组件通过通道 register 传递其标志符 i_d 和服务 s 到连接件中. 在连接件中相应的注册进程为

$$\text{creginfo}(x,y)=(x,y)(\text{register}(x,y))$$

表示连接件通过通道 register 获取新组件的注册信息 x 和 y，同时更新其知识库 K_p.

当新组件请求一个服务的时候，需要通过连接件查找提供相应服务的组件. 如果存在相应的服务组件，连接件负责建立这两个组件之间的交互通道；如果不存在相应的服务组件，请求组件可以向连接

件定购这个服务.新组件请求服务的进程为

$$\text{requestservice}(i,r,l)=\bar{i}\langle a\rangle.r(z).\bar{z}\langle l\rangle$$

表示新组件通过通道 i 发送请求 a 到连接件查询相应的服务组件,然后通过通道 r 等待连接件的答复.一旦新组件收到了服务组件的标志符 z,就通过 z 发送请求服务的地址 l 到服务组件.

在连接件中相应的服务查询进程为

$$\text{queryservice}(i,r,p)=i(y).([y=a]\bar{r}\langle p\rangle+[y\neq s]\overline{\text{subscribe}}\langle y\rangle)$$

表示连接件通过通道 i 接收请求组件的请求 y,然后通过知识库判断是否存在相应的服务 a,如果存在 a,则通过通道 r 发送该组件的标志符 p;否则,定购该服务.

在服务组件中相应的服务提供进程为

$$\text{povdservice}(p,s)=p(x).\bar{x}\langle s\rangle$$

表示服务组件通过通道 p 接收服务请求的地址 x,并通过 x 发送相应的服务 s 到请求组件.

根据上述的 π 演算进程可以建立请求组件和服务组件之间的交互通道.

如果系统体系结构发生变化或组件升级,就需要对原组件进行更新.组件的更新方式分为两种情况:①由于组件内部出现错误或算法效率低,需要对组件内部进行更新,即组件的内部演化,称为保持语义更新;②由于组件新版本的出现,对外界提供了一些新的功能,即组件内部实现和接口都发生了演化,称为扩展性更新.对于第一种情况,可以利用 π 演算的弱等价关系判断新旧组件能否进行更新.更新规则如下.

规则 1　设组件 $C_{p,0}$ 具有接口 P,组件 $C_{p,1}$ 具有接口 Q,若 $P\approx Q$,则组件 $C_{p,1}$ 可以时 $C_{p,0}$ 进行保持语义更新.

对于第二种情况,由于新组件提供新的功能,组件的原有接口发生变化或提供新的接口,在进行组件更新时,可能新旧组件的行为并不完全等价[6].下面给出组件更新规则.

规则 2　设组件 $C_{p,0}$ 具有接口 P,组件 $C_{p,1}$ 具有接口 Q,P 和 Q 满足以下条件:

(1) $f_n(P)\subseteq f_n(Q)$

(2)如果 $P\xrightarrow{\tau}P'$,则 $\exists Q',Q\Rightarrow Q'$

(3)如果 $P\xrightarrow{x(z)}P'$,那么 $\exists Q',Q\xrightarrow{x(z)\cdots x_i(z_i)}Q'$ ($\xrightarrow{x(z)\cdots x_i(z_i)}$ 表示 Q 可响应若干个输入行为)

(4)如果 $P\xrightarrow{\bar{x}\langle y\rangle}P'$,那么 $\exists Q',Q\xrightarrow{\bar{x}\langle y\rangle\cdots\bar{x}_i\langle y_i\rangle}Q'$ ($\xrightarrow{\bar{x}\langle y\rangle\cdots\bar{x}_i\langle y_i\rangle}$ 表示 Q 可执行若干个输出行为)

则进程 Q 对 P 进行扩展性更新,记为 $P\prec Q$,那么组件 $C_{p,1}$ 对 $C_{p,0}$ 进行了扩展性更新.

条件(1)说明,P 的自由名集合为 Q 的自由名集合的子集,条件(2)和(3)分别说明,Q 除了响应和执行 P 的动作外,还可以执行其他的动作.

在分布式应用中,为了提高系统性能而配备新的服务器,需要把客户端的请求平均分配到不同的服务器上,对负载进行均衡,借此缩短系统的响应时间.在 SAAM 中,由于通过连接件建立组件之间的交互通道,可以通过连接件对服务组件的负载进行调整.假如系统中有两个服务组件,随着时间的变化,系统中客户端的数量越来越多,可以通过连接件调整客户端的请求使两台服务器的负载保持平衡,调整规则如下

$$\begin{aligned}&\text{if } S_1.n\leqslant S_2.n\\&\text{then}\quad \text{queryservice}(i_1,r_1,p_1)\\&\text{else}\quad \text{queryservice}(i_2,r_2,p_2)\end{aligned}$$

这个规则表示连接件在建立客户端和服务器的交互通道时,首先判断两台服务器 S_1 和 S_2 中客户端的数量,如果第一台服务器的客户数量少于第二台,则通过连接件的服务查询进程发送第一台服务器的标志符给顾客,从而建立它们之间的交互通道;反之,建立客户端与第二台服务器的交互通道.

2.2　软件体系结构一致性分析

在软件系统运行时,由于新组件的加入或组件的更新,导致软件体系结构发生演化,这可能会影响系统的一致性.一致性是指软件体系结构中各个部分必须成功的进行交互,而不会彼此冲突.动态演化必须保证系统体系结构的一致性[7].

在 SAAM 中,组件把相应的信息注册在连接件中,通过连接件建立组件之间的交互通道,然后组件按照连接件规定的协议进行交互.因此,体系结构的一致性主要指组件之间交互的一致性.由于组件通过接口进行交互,而且组件接口与内部实现是兼容的,因而可以在接口层次上判断组件交互的一致性[8].

如果两个进程 P 和 Q 是一致的,则 P 和 Q 至少能借助一个共享的对偶名字进行交互,如 $P=\bar{x}\langle y\rangle.P'$,$Q=x(z).Q'$,这样的进程之间存在同步关系.

定义 2　进程半一致性[6]　设同步进程集合上的二元关系 R,对于 PRQ,且对所有的替代 $\sigma\notin f_n(P)\cup$

$f_n(Q)$（f_n 为进程的自由名集合），满足 $P\sigma RQ\sigma$，而且下列条件成立：

(1)如果 $P\xrightarrow{\tau}P'$，则 $P'RQ$

(2)如果 $\neg(P\sim 0)\wedge\neg(P\equiv 0)$，$P\xrightarrow{x(z)}P'$，$Q\xrightarrow{\bar{x}y}Q'$，则 $P'\{y/z\}RQ'$

(3)如果 $\neg(P\sim 0)\wedge\neg(P\equiv 0)$，$P\xrightarrow{x(n)}P'$，$Q\xrightarrow{\bar{x}(n)}Q'$，则 $P'RQ'$

则称 R 为进程间半一致性关系. 如果 R 和 R^{-1} 都是半一致性关系，则称 R 为进程间一致性关系，记为 $\backsim$.

进程的一致性确保在进程交互中不存在不匹配的情况，强调进程的正常交互，表明进程能够执行内部演化而成功结束. 如果存在进程 P'，使得 $P\Rightarrow P'$，且 $P'\xrightarrow{\tau}$ 或 $P'\equiv 0$（$P'\xrightarrow{\tau}$ 表示 $\exists P''$，$P'\xrightarrow{\tau}P''$），则进程 P 可正常运行；如果 $P\Rightarrow P'$，$\neg(P'\xrightarrow{\tau})$ 且 $\neg(P'\equiv 0)$，就表示进程死锁，进程不能再执行内部演化，其行为也并未完全执行.

定理 1 设 P 和 Q 为一致性进程 $P\backsim Q$，如果组件 $C_{p,1}$ 和 $C_{p,2}$ 能分别由 P 和 Q 正确描述，则组件 $C_{p,1}$ 和 $C_{p,2}$ 满足一致性，即 $C_{p,1}\mid C_{p,2}$ 能正常交互.

由于组件接口描述组件内部实现，根据定义 2，利用数学归纳法可以证明定理 1 成立. 定理 1 保证了组件行为的匹配性.

定理 2 如果软件系统中组件完全满足一致性，则该系统可以正常交互.

定理 2 的证明可由定理 1 导出，证明比较简单，这里从略.

尽管在动态体系结构分析阶段已分析了组件的演化及系统的一致性，但局部的一致性分析并不能确保系统的无死锁性，还要利用 π 演算的相关分析方法对最终模型的死锁等性质进行分析和验证.

3 自动加油站系统建模与分析

考虑一个在软件体系结构领域广泛研究的实例——自动加油站系统. 加油站中包括顾客、出纳和加油泵，假设加油站中有充足的油料而且出纳不需要找零.

系统初始时，加油站中只包括一个顾客、一个出纳和一个加油泵. 我们可以把这个系统抽象为顾客 1、出纳和加油泵组件，并包括一个连接件，其初始 SAAM 配置如图 2 所示，组件与连接件的接口用带阴影的变迁表示，且带有卫函数. 为了模型简洁，我们忽略了数据类型、弧表达式和卫函数. 用 Wright 对自动加油站进行建模[9]，需要 3 个组件、3 个连接件和 12 个附件，建模比较繁琐. 利用 Darwin 建模时[10]，系统中没有连接件，顾客组件利用相同的接口付钱和加油，容易造成冲突，体系结构配置比较混乱，没有正确地对体系结构进行抽象. 相比之下，SAAM 简单、直观易懂，能保证组件的正确交互. 在 SAAM 中，顾客、出纳和加油泵组件首先在连接件中注册并查找相应的服务组件，连接件为它们建立 pay_1、$pump_1$ 和 info 消息传递通道，然后各个组件与相应的服务组件进行交互.

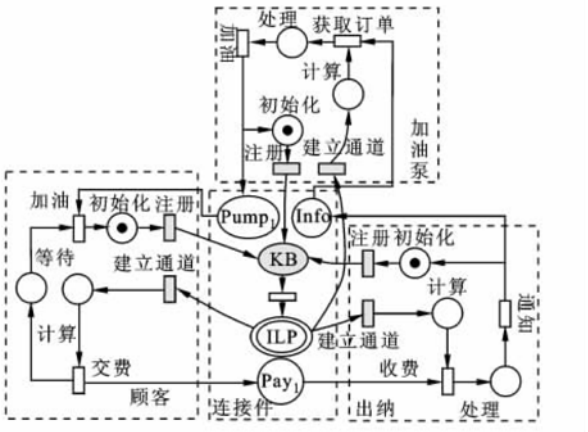

图 2 自动加油站初始 SAAM

可以利用 Petri 网支持工具 INA 分析系统性质. 根据 INA 分析结果，图 2 所示的初始 SAAM 是活的，不存在死锁，可以保证系统的正常运行.

在系统运行中，加油站中不断有新的顾客到达，系统结构呈现动态变化. 假设某时刻顾客 2 到达加油站，则根据 2.1 节中的连接件，利用 π 演算建立组件之间交互通道的动态过程. 顾客 2 首先在连接件中注册，随后建立与出纳、加油泵之间的交互通道 pay_2 和 $pump_2$，此时系统中包含 2 个顾客，系统体系结构发生变化. 我们现在要关注的是顾客 2 到达后系统的一致性问题，即系统是否能正常交互. 因此，我们首先根据 2.2 节的体系结构一致性分析，把新加入组件的接口和与新组件有交互行为的组件接口用进程描述出来，然后根据接口判断新组件的加入是否会影响系统的一致性. 3 个组件的接口可以定义如下：

顾客 2（$C_{p,cus2}$）：$pay(m)=\overline{pay_2}\langle m\rangle.\,pay(m)$，$inputgas(x)=pump_2(x).\,imputgas(x)$；

出纳（$C_{p,cas}$）：$charge(y)=pay_2(y).\,charge(y)$，

inform(i_d,m_{sg})=$\overline{info}\langle i_d, m_{sg}\rangle$. inform($i_d$,$m_{sg}$);

加油泵($C_{p,pum}$):pump(g)=$\overline{pump_2}\langle g\rangle$. pump($g$),getorder($z$,$w$)=info($z$,$w$). getinfo($z$,$w$).

出纳的接口 inform(i_d, m_{sg})根据顾客的 i_d 通知加油泵给相应的顾客加油,根据定义 2 很容易得到 pay (m) ∽ charge (y), inform (i_d, m_{sg}) ∽ getorder(z,w),inputgas(x)∽pump(g),而且顾客 1 与出纳和加油泵的交互通道与顾客 2 的不相同,不会发生冲突. 因此,根据定理 1,由这些接口描述的组件 $C_{p,cus2}|C_{p,cas}|C_{p,pum}$ 能正常运行,进而根据定理 2,由这些组件组成的系统能保持正常交互. 系统发生动态变化后,其最终模型用 π 演算可以描述为

$$C_{p,cus1} = \overline{pay_1}\langle m_1\rangle.\ pump_1(x_1).C_{p,cus1}$$

$$C_{p,cus2} = \overline{pay_2}\langle m_2\rangle.\ pump_2(x_2).C_{p,cus2}$$

$$C_{p,cas} = pay_1(u).\overline{info}\langle i_{d_1}, m_{sg_1}\rangle.C_{p,cas} + pay_2(v).\overline{info}\langle i_{d_2}, m_{sg_2}\rangle.C_{p,cas}$$

$$C_{p,pum} = info(z,w).([z = i_{d_1}]\overline{pump_1}\langle w\rangle + [z = i_{d_2}]\overline{pump_2}\langle w\rangle).C_{p,cas}$$

$$S_M = C_{p,cus1} \mid C_{p,cus2} \mid C_{p,cas} \mid C_{p,pum}$$

最后利用 π 演算的支持工具 MWB 分析最终的 SAAM. 分析结果表明,最终的模型不存在死锁,系统可以正常运行.

4　结束语

本文针对用现有方法对动态体系结构建模的不足,以两种互为补充的形式化方法——面向对象 Petri 网和 π 演算为基础,建立了软件体系结构抽象模型 SAAM. 面向对象 Petri 网作为一种具有严格数学语义的形式化图形建模工具,可以比较形象地描述软件体系结构的初始化模型及动态行为,而 π 演算能够描述动态演化的系统结构. 利用 OPN 与 π 演算的支持工具,可以对 SAAM 进行分析和验证. 最后把 SAAM 应用于加油站系统的建模与分析,结果表明 SAAM 比较直观、易懂,具有较好的工程应用前景.

参考文献:

[1] Oquendo F. π-ADL: an architecture description language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures [J]. ACM Software Engineering Notes, 2004, 29 (4): 1-13.

[2] He Xudong, Yu Huiqun, Shi Tianjun, et al. Formally analyzing software architectural specifications using SAM [J]. Journal of Systems and Software, 2004, 71 (1/2): 11-29.

[3] Medridovic N, Taylor R N. A classification and comparison framework for software architecture description languages [J]. IEEE Transactions on software Engineering ,2000,26(1):70-93

[4] 于振华, 蔡远利. 基于面向对象 Petri 网的软件体系结构描述语言 [J]. 西安交通大学学报, 2004, 38(12): 1236-1239.
Yu Zhenhua, Cai Yuanli. Software architecture description language based on object-oriented Petri nets [J]. Journal of Xi′an Jiaotong University, 2004, 38 (12): 1236-1239.

[5] Milner R, Parrow J, Walker D. A calculus of mobile processes [J]. Journal of Information and Computation, 1992, 100(1): 1-77.

[6] Canal C, Pimentel E, Troya J M. Compatibility and inheritance in software architectures [J]. Science of Computer Programming, 2001, 41: 105-138.

[7] Goudarzi K. Consistency preserving dynamic reconfiguration of distributed systems [D]. London: Imperial College London, 1998.

[8] Cimpan S, Leymonerie F, Flavio O F. Handling dynamic behaviour in software architectures [M]// Lecture Notes in Computer Science. Berlin: Springer, 2005: 77-93.

[9] Naumovich G, Avrunin G S, Clarke L A, et al. Applying static analysis to software architectures [M]// Lecture Notes in Computer Science. New York: Springer-Verlag, 1997: 77-93.

[10] Magee J, Kramer J, Giannakopoulou D. Behaviour analysis of software architectures [C]// Donohoe P. Proceedings of the 1st Working IFIP Conference on Software Architecture. Boston: Kluwer Academic Publishers, 1999:35-50.

(编辑　刘　杨)