软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2012,23(4):802−815 [doi: 10.3724/SP.J.1001.2012.04029]

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

自主构件自适应策略的在线定制及动态评估*

接钧靖[1,2], 史庭训[1,2], 焦文品[1,2+], 孟繁晶[3]

[1](北京大学 信息科学技术学院 软件研究所,北京 100871)

[2](高可信软件技术教育部重点实验室(北京大学),北京 100871)

[3](IBM 中国研究院,北京 100193)

Autonomous Components Whose Adaptation Policies Can Be Customized Online and Evaluated Dynamically

JIE Jun-Jing[1,2], SHI Ting-Xun[1,2], JIAO Wen-Pin[1,2+], MENG Fan-Jing[3]

[1](Institute of Software, School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China)

[2](Key Laboratory of High Confidence Software Technologies of Ministry of Education (Peking University), Beijing 100871, China)

[3](China Research Laboratory, IBM, Beijing 100193, China)

+ Corresponding author: E-mail: jwp@sei.pku.edu.cn

Jie JJ, Shi TX, Jiao WP, Meng FJ. Autonomous components whose adaptation policies can be customized online and evaluated dynamically. *Journal of Software*, 2012,23(4):802−815. http://www.jos.org.cn/1000-9825/4029.htm

Abstract: Self-Adaptive software in open and distributed environments (especially the Internet) has been widely researched in academia and industry. However, software entities scattered on the Internet are independently developed and deployed by different organizations, and they autonomously take actions on behalf of their owners. They can no longer be considered passive and manageable. In the construction of self-adaptive software systems in open and distributed environments, constituent elements should be modeled and designed as autonomous computing entities, and the adaptive logic of systems should be encapsulated into constituent elements. Existing researches on autonomous computing entities are still insufficient in self-adaptive policies' in online customization and dynamic evolution. Therefore, this paper proposes an autonomous component model, which supports self-adaptive policies' in online customizing, by which components can gain new policies or behavior modes at runtime. Meanwhile, this paper implements a self-adaptive mechanism based on dynamic quality evaluation, by which components can evaluate the policies and select the best policy to improve their qualities of service at runtime. Finally, the paper provides some implementation details of the proposal and an experiment, which demonstrates the process of self-adaptation based on dynamic quality evaluation and the process of online policy customization.

Key words: autonomous component; self-adaptive; quality; dynamic evolution

摘 要: 开放分布式环境下自适应软件的研究已引起学术界、工业界的广泛关注.但分布在网络上的软件实体是由不同的组织独立开发并部署的,它们代表各自的组织(或所有者)自主地采取行动,在构造分布式环境下的自适应

* 基金项目: 国家自然科学基金(61073020); 国家重点基础研究发展计划(973)(2009CB320703); 国家高技术研究发展计划(863)(2008AA01Z139)

收稿时间: 2011-01-19; 定稿时间: 2011-03-21

系统时,不能再将构成单元视为被动的受管对象,而应将其建模为具有主动行为能力的计算实体,并在这一层面设计和封装系统的自适应逻辑.然而,在现阶段对于自主计算实体的研究中,大多缺乏对于自适应策略的动态加载和动态演化的支持.提出了一种支持策略动态加载的自主构件模型,使得自主构件能够在运行时习得新的自适应策略和行为,实现了一种基于质量运行时动态评估的自主构件的自适应机制,使得自主构件能够自行评估自适应策略的优劣并选择最佳的策略加以适应,在保证自身目标得以实现的同时,提高了服务质量.另外,还详细描述了自主构件的实现方案及其运行支撑,通过实验展示了自主构件基于质量动态评估的自适应过程以及自适应策略的动态加载过程.

关键词: 自主构件;自适应;质量;动态演化

中图法分类号: TP311 **文献标识码**: A

在过去的 20 多年中,IT 工业获得了空前的发展:各种计算设备大量增长,计算能力飞速提高,信息呈爆炸式增长,新技术层出不穷.同时,信息技术与日常工作和生活日渐融合,个人、企业、社会越来越依赖于信息系统.然而,伴随着这种繁荣的态势,日益增加的软件复杂性也已成为一个潜在的巨大危机,主要表现在如下 3 个方面:1) 软件越来越复杂,基于传统的管理方式将难免导致软件维护人员的大量增加,从而大大加重了企业的信息化成本;2) 在多样化的运行环境中,例如普适计算和移动计算等,都要求软件对终端用户尽可能地透明;3) 在许多应用领域,软件的离线调整往往会造成较大的系统开销,所以人们需要新的技术手段,支持在运行过程中对这些应用进行调整[1].

上述 3 个方面的特征迫切要求软件自身能够感知并适应环境,如果软件能够在运行时观察其所处的环境,对自身的状态及计算行为做出调整,将大大降低软件维护人员的工作量,并消除离线调整带来的开销.在这一背景下,对于自适应软件的研究引起了学术界、工业界的广泛关注.如产业界中 IBM 发起的自治计算(autonomic computing)、微软的动态系统(dynamic system);学术界中提出的自适应软件(self-adaptive software)、自组织系统(self-organizing system)等等.

另一方面,随着网络技术的持续发展以及面向服务的计算、普适计算、网格计算等新计算范型的不断涌现,Internet 已经由内容服务的提供者转变成了计算服务的运行平台,越来越多的软件系统直接通过组装分布在 Internet 上的计算资源(如 agent,Web service,COTS 等软件实体)来实现.但是,分布在网络上的软件实体是由不同的组织开发并部署的,它们代表各自的组织(或所有者)采取行动.这些软件实体往往是自主的,它们会在不通知他人的情况下主动地进入或退出环境,并且不断地演化.它们又是相互独立的,软件系统开发人员并不拥有它们,它们的行为可能不受软件系统开发人员的控制,它们的服务质量不仅仅取决于它们自身,还有赖于它们所部署和运行的环境.因此,它们的行为和服务质量可能是不确定的和无法预知的.现有的自适应软件构造方法往往不能很好地直接应用于软件系统的构造,其主要问题可归纳为以下 3 个方面:1) 软件系统中较难建立准确的全局模型;2) Internet 环境的不稳定性使得基于集中控制的系统存在单点失效的隐患;3) 软件系统的开放性和构成成分的独立性提高了自适应机制的复杂程度.

一条解决途径是在构造分布式环境下的自适应系统时,不再将运行系统中的构成单元视为被动的受管对象,而是将其建模为具有主动行为的计算实体,在这一层面设计和封装系统的自适应逻辑.近年来,研究如何在分布的计算实体中封装自主策略,通过它们之间的交互来满足系统的自适应需求,已被本领域不少知名研究人员认为是实现 Internet 环境下软件自适应的重要课题之一[2].由于软件 Agent 被认为是具有自主性的软件系统,而且 FIPA 组织制定了一系列 Agent 之间以及 Agent 与其他软件系统之间协同、交互的标准[3],因此,大多数比较有影响的自主计算实体模型都是通过向传统的计算实体中加入一个负责处理自适应逻辑和交互逻辑的 Agent 来实现的,例如自管理系统[4]、Fractal[5]、Accord[6]、*K*-component[7].然而,这些自适应逻辑都是在设计阶段预先设定好的,不具备学习能力,而且大多数也不支持策略的在线更新,对于复杂的系统,设计者无法制定出一套完美的自适应策略,只能不断重复测试-重新开发-测试的循环;另一方面,这些自适应逻辑具有很强的应用相关性,这导致了对其进行维护和扩展将会带来较大的开销.

因此,本文提出了一种支持策略动态定制和运行时动态评估的自主构件模型,并实现了其运行支撑.该自主

构件模型包括 3 个主要模块,分别是功能模块、策略模块和质量模块.支持两种自适应机制,分别是基于规则的自适应和基于质量动态评估的自适应.基于规则的自适应由策略模块实现,就像人的反射弧,能够根据外部的环境和内部的状态进行快速的反射式调整.同时,设计者还可以在线加载新的策略;而基于质量动态评估的自适应则由质量模块实现,就像是人的大脑,能够在运行时刻获取并存储知识,并对影响质量的各个参数进行慎重而复杂的推理;至于功能模块,实际上是一个普通的构件,就像是人的肢体和器官,负责完成基本的功能逻辑.开发者只需分别设计功能模块、策略模块和质量模块,自主构件运行支撑将会自动地将 3 部分联系起来.

本文第 1 节对软件的自适应进行讨论.第 2 节对自主构件的各个模块进行简单介绍.第 3 节详细介绍自主构件策略的在线定制以及如何在运行时进行基于质量动态评估的自适应.第 4 节描述一个应用实例,给出如何用我们的自主构件模型实现该场景、如何对策略进行动态评估以及如何动态定制新的策略,并对实验结果进行分析讨论.第 5 节比较相关的研究工作.第 6 节总结全文并给出进一步的研究计划.

1 软件的自适应

在介绍我们的构件模型之前,首先讨论一下软件的自适应.我们提出了一种关于自适应中自评估过程的新的理解,这也是我们的构件模型中基于质量的动态评估的理论基础.

对于什么是自适应软件,其中最早被广泛接受的是 MIT 人工智能实验室的 Laddaga 于 1997 年末提出的定义[8]:

“自适应软件评估自身的行为,当评估显示其自身并非正接近所倾向完成的目标,或其有可能提供更好的功能或性能时,软件改变自身的行为”.

一般认为,一个具有一般性的完整的自适应过程应该包括采集-分析-决策-执行这 4 步[9].关于信息的采集,已有很多技术上的支持,包括软件的和硬件的,但这些不在本文的讨论范围之内.关于自适应的执行,主要包括调整构件内部属性值(构件状态)、选择所依赖服务的提供者、改变所提供服务的实现方式、与用户或其他构件之间进行交互等.本文着重讨论分析和决策过程在自主构件中的实现,而在这两个步骤中,最关键的问题就是构件质量的自评估.

构件质量自评估的基本思想是,列出影响自主构件质量的所有因素,这些因素包括可控的(如构件内部属性值、构件对于外部服务提供者的选择、构件所提供服务的实现方式等)和不可控的(如外部环境信息等).在运行时刻获得并动态更新这些因素的不同取值与构件质量之间的关系.这样就能够在检测到不可控因素的取值之后,调整可控因素的值,使得构件的质量始终保持在一个较高的水平.因此,质量评估以及决策的核心是:1) 如何找出这些影响因素;2) 如何根据运行时刻动态收集到的信息对当前的自适应决策(各影响因素的取值)进行评价(即自评估);3) 如何根据已有的评价做出新的自适应决策.

Laddaga 在文献[8]中根据所能获得的知识的多少,将自评估分为:构造性的(constructive)、有确切依据的分析(analytic with ground truth)和没有确切依据的分析(analytic without ground truth).举例来说,一个机器人要抵达一个目的地,如果它掌握了关于所有障碍物的全局知识,能够据此事先评估并规划出一条最优路径,则认为这种评估是构造性的;如果不能事先获得全局信息,只能在移动过程中根据被告知的离目的地的距离,对路径进行评估,则认为这种评估是有确切依据的分析;如果连离目的地距离的知识也无法获得,这种评估是最难的,叫做没有确切依据的分析.在这种情况下,可以通过自进化(如遗传算法)来达到自适应的目的.

事实上,构造性的评估是指在做出新的自适应决策之前就能够获得评估其所有自适应决策的信息,有确切依据的分析是在自适应决策做出之后才能收集到评估该决策所需要的信息,而没有确切依据的分析则是自适应个体的生命期到达了某个特定的阶段(或完成了某个完整的功能)之后才能够获得评估其自适应决策所需要的信息.可见,“时间”在自适应的过程中是一个很重要的因素.最糟糕的自适应方式是,将要自适应的个体复制很多份,每份拥有不同的配置,在系统运行的过程中,那些质量较差的个体将自然地被淘汰,这种自适应方式实际上是在自身完成其功能之后才进行自评估;较好一点的自适应方式是,自适应的个体时刻监控着外部的反馈信息,每做出一个自适应决策,都根据外部的反馈信息修正自身的行为,这种自适应方式实际上是在自适应决策之

后可接受的时间范围内对自身进行自评估;最理想的自适应方式是,自适应的个体拥有丰富的经验,无论遇到什么情况,他都能够根据自己的经验做出合理的选择,这种自适应方式实际上是在他做出自适应决策之前就进行自评估.所以,自评估的时间越早,自适应的效率就越高.一些研究工作通过建立预知系统,分析系统运行的历史信息来预知系统的行为,做出及时调整以防止系统崩溃[10],实际上就是将系统自评估的时间提前.

在经过上述讨论之后,需要解决的问题是:如何将自评估的时间提前.我们认为,系统的质量属性是一个与时间有关的变量:$\langle Q,t\rangle$.例如:开发者希望系统的平均响应时间少于1s,那么$\langle Q,t\rangle$中的Q就代表系统从稳定运行到时刻t这一段时间内的平均响应时间,而t则代表系统下一次停机的时刻.自适应的目的是使得$\langle Q,t\rangle$的值小于1s.而要评估一个自适应操作是否达到了我们所期望的目的,需要获得系统从稳定运行到下一次停机这段时间内所有响应时间的平均值.显然,t 的取值越大越准确,但这同时又会导致自适应缺乏效率,因此有必要将这个自评估的时间提前.如果我们假设当系统做出自适应调整之后100s进入稳定状态,并从此之后每10s测试一次响应时间,取样10次,那么这个变量可以分解为$\langle Q,t\rangle=(\langle R,110\rangle+\langle R,120\rangle+\ldots+\langle R,200\rangle)/10$,其中,$\langle R,t\rangle$表示系统在$t$时刻的响应时间.这样,我们就将系统的自评估时间提前到了系统做出自适应调整之后的第 200s.根据之前的讨论,如果我们能够预先确定做出某个自适应操作之后$\langle Q,t\rangle$一定小于 1s,那么自评估就是构造性的.如果构件平均每300s被调用1次,那么200s的自评估时间是可以接受的,自评估就是有确切依据的分析.如果构件每100s被调用1次,试用期的高响应时间会给调用者带来不好的印象,那么就是没有确切依据的分析.此时,我们需要将自评估时间进一步提前.例如,如果我们能够确定系统响应时间的波动幅度在0.2s之内,那么从110s~200s这段时间的10个采样点的平均值将会在$\langle R,110\rangle+0.18$s~$\langle R,110\rangle-0.18$s这个范围内,因此我们可以用$\langle R,110\rangle$来估计系统的平均响应时间,尽管精确度下降了.上面所描述的例子只是一种简单的自评估情况,这个例子说明,自评估的本质是要将评估的时机提前到一个合理的范围内,而在实际应用中又往往需要考虑自适应效率与精确性的折衷.

对于复杂的情况,系统的自评估往往需要考虑到很多变量.例如,如果系统响应时间主要受自身的m操作和n操作影响,而 n 操作是调用外部服务,受网络环境 e 和外部服务 k 影响,那么$\langle Q,t\rangle=\langle Rm,t\rangle+\langle Re,t\rangle+\langle Rk,t\rangle$,其中,$\langle Rm,t\rangle$表示 m 操作的平均响应时间,$\langle Re,t\rangle$表示网络环境的平均延迟,$\langle Rk,t\rangle$表示外部服务 k 的平均响应时间.$\langle Rm,t\rangle$和$\langle Rk,t\rangle$为可控的($\langle Rk,t\rangle$可以通过选用不同的外部服务来调节),$\langle Re,t\rangle$为不可控的.并且,由于这3个变量不是独立的,它们之间的相互影响也使得自评估变得更加复杂.理想状态下,我们希望得到$\langle Rm,t\rangle$,$\langle Re,t\rangle$,$\langle Rk,t\rangle$在不同的取值下系统的质量属性$\langle Q,t\rangle$的值.也就是说,自评估的理想结果是得到所有变量在所有可能的取值下,系统质量属性的取值表.而随着变量的增多以及变量的可能取值的增多,取值表的空间将以指数级增长,尝试所有的变量取值组合将带来很大的代价,这也导致了自适应效率的降低.对于因影响因素过多而导致自适应效率降低的问题,主要有3种解决办法,如图1所示.

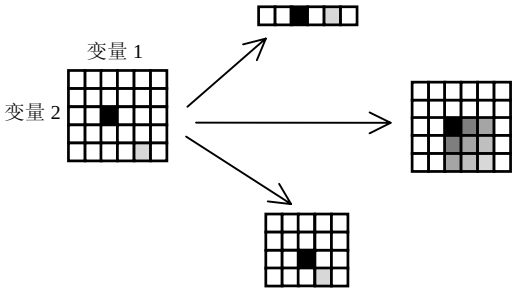


Fig.1 Efficiency problem caused by too many related variables

图1 影响因素过多带来的自适应效率问题

图1左侧表示构件的质量取值表,主要受两个因素(变量1、变量2)的影响,并且已经对其两组取值进行了评估(深颜色代表质量较高,浅颜色代表质量较低).如果要填满整个表,总共需要进行 5×6=30 次取样评估.因此,我们可以通过放弃考虑不重要的变量(如变量 2)来提高评估的效率(中间偏上图),或者丢弃不太可能的变量取值(中间偏下图),或者通过某种算法来估算未评估点的取值(右侧图)来提高自适应的效率.

综上所述,在实现质量自评估时,需要考虑的两个重要的问题是:1) 如何通过将自评估时间提前来提高自适应的效率;2) 如何解决由于影响因素过多导致的自适应效率问题.

2 自主构件模型及原理

与传统的构件相比,自主构件除了是独立开发的并能被第三方复用以外,它还表现出对自身行为的控制能力和实现目标的决策能力,即它能在不需要外界(包括人)干预的情况下,自行决定如何行动以及如何实现自身的目标.在我们看来,构件具有(或表现出)自主性的目的是为了自适应,即能够在外界环境发生变化时,自己决定如何适应环境的变化,尽可能地以最佳的方式(或服务质量)实现目标.

尽管在软件的自适应中一般都会涉及到质量问题,但目前在谈到自适应机制时,一般都是基于规则的,因为基于规则的自适应相对来说比较简洁,也容易实现.但对于复杂的自适应问题,基于规则的自适应存在很多缺陷,比如:

1) 基于规则的自适应是基于事件触发的、被动的、同步的自适应.除了这种同步的自适应以外,自主构件还应具有主动的、异步的检查和修正自身偏差的能力;
2) 基于规则的自适应抽象层次较低,表达能力有限,不适用于复杂的自适应问题.对于稍微复杂的系统,规则的编写都是很难的.比如,规则的前件是某个复杂处理的结果或者包含很多参数;又如,很难在系统设计阶段就制定出完美的规则;并且由于环境的动态改变,也不可能存在一成不变的完美规则;而规则本身又不能在运行过程中动态演化.这些都给开发者带来很大的困难;
3) 涉及到规则的冲突、优先级等问题,基于规则的系统不利于维护,扩展性差.当需要添加新的规则时,检测新规则与旧规则的冲突等问题会带来很大的开销.

因此,我们提出一种与规则机制优劣互补的自适应机制,称为质量动态评估的自适应机制.基于规则的自适应能够根据外部的环境和内部的状态进行快速而简单的反射式调整;而基于质量动态评估的自适应则能够在运行时刻获取并存储知识,并对影响质量的各个参数进行慎重而复杂的推理.同时,二者在为开发者提供便利方面又有着相似之处.基于规则的自适应使得开发者将注意力集中在规则的编写上,将自适应与业务逻辑分离开来,规则引擎将负责规则的切实执行;而基于质量动态评估的自适应则使得开发者将注意力集中在对影响自主构件质量的因素的管理以及对质量的评估上.两种自适应机制优劣互补,将使得自适应逻辑更清晰,能力更强.

2.1 自主构件的模型

我们的自主构件模型是在普通的构件模型的基础上进行扩展,加入了策略模块和质量模块,如图 2 所示.

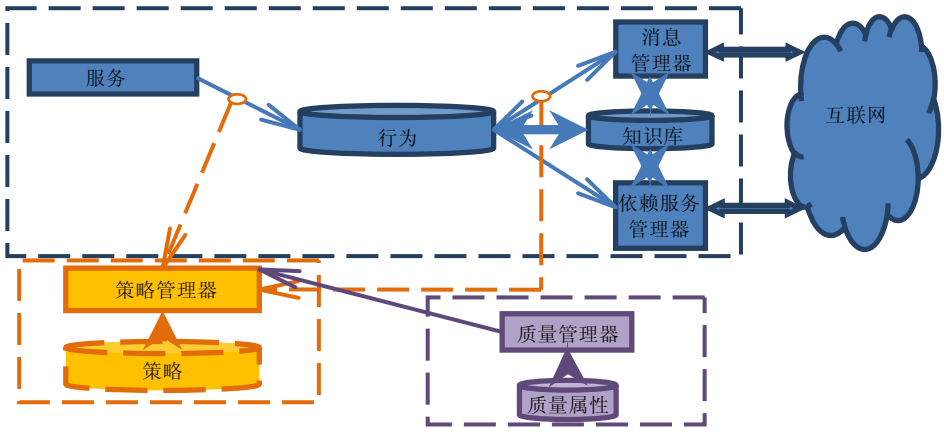


Fig.2 Model of autonomic component

图 2 自主构件模型

2.1.1 功能模块

功能模块实际上是一个普通构件,一个没有实现策略模块和质量模块的自主构件可以作为普通构件在J2EE 服务器上运行.功能模块主要包括服务(service)、消息管理器(message manager)、依赖服务管理器(dependency service manager)、行为(behavior)和知识(knowledge)这 5 个部分.服务就是构件对外提供的服务,消息管理器和依赖服务管理器分别通过消息或者服务调用的方式实现构件与外部的交互.消息管理器和依赖服务管理器所需要用到的关于外部交互对象的信息都保存在知识库中.另外,知识还包括构件的内部状态等信息.自主构件采用了双向选择的机制选择外部的消息交互对象或服务提供者,并在运行支撑中实现了一系列 API 对该机制提供支持.我们在文献[11]中对自主构件的双向选择机制进行了较为详细的介绍.构件对外提供的服务以及对于消息的处理都是由行为实现的.行为可以分为很多种,例如:实现构件所提供的服务的行为,称为服务实现行为;处理消息的行为,由于很多是根据所接收到的信息进行一些自适应的调整,所以称其为自适应行为.同时,一个行为可能调用其他的行为,所以从这个角度又可以分为复合的行为和原子行为.实际上,开发功能模块与开发一个普通构件没有太大的区别.因此,我们将在未来的工作中研究如何在不对一个普通构件进行太多改动的前提下将其改造为一个自主构件.

2.1.2 策略模块

首先介绍什么是策略,由于规则有很多局限,例如大量的规则管理起来很麻烦,有的规则之间存在依赖关系、而有的规则之间是冲突的,这使得规则集的维护和复用变得十分困难.因此,我们提出了策略的概念以便对规则进行管理.一个策略描述了构件的某个功能或功能片段的实现方式,并且它关注于系统的某项非功能属性.系统可以选择不同的策略,使得系统所关注的非功能属性变得不同.我们用 Drools 规则流[12]的形式表示一个策略,规则流可以理解为工作流与规则的结合体,规则包含在规则流中的规则组里,可以在工作流执行的特定阶段触发规则检查.如图 3 所示,动作执行之后将触发规则组内的规则进行检查.

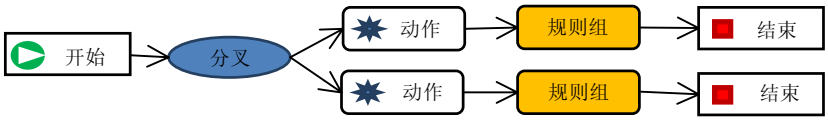


Fig.3 A sample rule flow

图 3 规则流示意图

实际上,每一个策略在能力上等价于一种复合行为.也就是说,我们可以将一个策略实现为构件的一种复合行为,用该复合行为的执行来代替策略的执行;也可以通过加入新的策略来替换或者扩展原有的复合行为.所以,与行为类似,策略也可以分为服务实现策略和自适应策略.加入策略的一个好处是,可以方便地对构件进行运行时扩展,同时,现有的基于规则的自适应构件中的规则也都可以实现为我们自主构件中的自适应策略.我们在以前的工作中研究了这种基于规则的自主构件运行支撑框架[13],而在本文的自主构件模型中,不再直接通过规则来指导构件的自适应行为,而是由策略来完成构件的自适应.这样做的好处是,将不同策略中涉及到的规则以及同一策略的不同执行阶段所涉及到的规则分离开,使得逻辑更加清楚,规则的维护成本也大为降低.另外,我们的自主构件中的策略是可以在线动态定制的,当开发者发现有更好的服务实现策略或自适应策略时,可以离线编写、编译,在线加载.

2.1.3 质量模块

只有策略模块而没有质量模块的自主构件也可以部署运行,但只能像基于规则的自适应构件那样进行反射式的自适应,虽然可以进行在线策略更新,但对于复杂的系统,策略的设计本身就是一件非常困难的事情,而且由于环境的不确定性,也不可能存在一成不变的完美策略.

我们在第 1 节中对于软件的自适应进行了讨论,质量模块就是基于前面的分析,使得自主构件的自适应更具一般性.基于质量动态评估的自适应机制的基本思想如下所述:

开发者对系统所期望达到的目标进行显式的定义,称为质量属性定义.具体来说,主要包括以下 3 个方面:

1) 找出影响质量的各个因素及可能的取值.影响质量的因素包括外部因素和内部因素.根据第 1 节中对于自适应效率问题的讨论,我们必须要把影响因素的可取值缩小到一个可接受的范围内.因此我们规定:外部因素的取值必须是有限的集合,例如,网络丢包率在 0.0~1.0 之间,我们可以将其分为低、中、高 3 类,使其只有 3 个取值.内部因素可取的值则必须是对策略的选择,也就是对于影响质量的功能或者功能片段的实现方式的选择.例如,如果缓冲区的大小是影响构件质量属性的因素之一,那么就必须有倾向于设置较大缓冲区的策略和倾向于设置较小缓冲区的策略,两种不同的策略就构成了缓冲区大小这个内部因素的两种取值;

2) 给出运行时对质量进行动态评估的方法.所谓评估,就是对影响因素的各种取值组合给出评分.根据第 1 节中对于自适应的讨论,为了保证自适应效率及效果,需要选择合适的评估时机,必要时还需要根据已有的评估结果对未评估的取值组合进行估计;

3) 给出根据上面 2)中的评估结果进行自适应决策的方法.一般默认的决策方法是:根据特定的外部因素的取值,找出一种内部因素取值组合(策略组合),使得质量属性评分最高.但是由于还有很多其他因素需要考虑,实际的决策方法可能比较复杂.例如,一种策略组合在被选中之后,可能需要一段时间才能观测到其影响(即我们前面所说的评估时机).因此,在一种策略组合未被评估之前不能进行新的决策.

3 策略的在线定制及基于质量动态评估的自适应

3.1 策略的在线定制

关于自主构件策略的在线定制,我们主要是引用了 JBOSS 的 Drools 规则流引擎,并在此基础上进行了一些扩展,形成了我们的策略管理器.相关资料可以参考文献[12],这里只作简单介绍.

图 4 中灰色的表格是实现策略映射表(match table).例如,功能模块中某个服务 SampleService 的实现可能是由名字为 SampleStrategy 的策略描述的.但构件的策略库中可能有多个该服务的实现策略(即存在多个名字为 SampleStrategy 的脚本),这些不同的实现可能有的效率较高,有的安全性较强.但构件同一时刻只能选择其中一个作为其功能接口的实现者.实现策略映射表的作用就是维护策略名字与具体实现策略之间的映射,策略管理器提供了访问 Match Table 的各种接口.除此之外,图中的其他组成部分都是 Drools 规则流引擎的组成成分.其中两个比较关键的模块是 Knowledge Builder 和 Knowledge Base.Knowledge Builder 的作用是将描述性的策略脚本(图中的 resource)编译成 Knowledge Package,Knowledge Package 将被添加到自主构件的知识库(knowledge base)中,当要执行某个策略时,策略管理器先从 Match Table 中查到具体实现策略的 ID,并将该 ID 传给 Knowledge Base,然后,Knowledge Base 将创建该实现策略的执行实例 Knowledge Session.

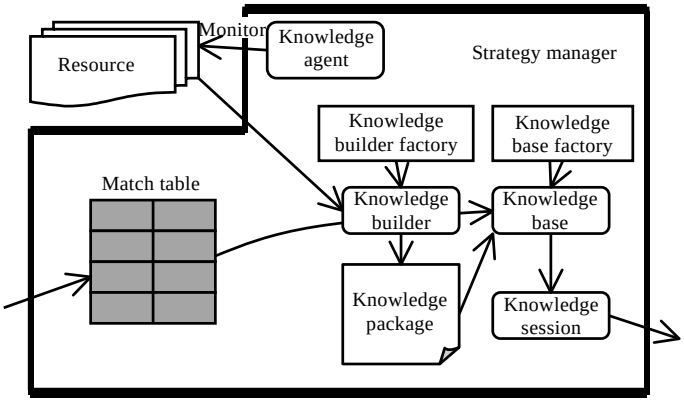


Fig.4 Strategy manager

图 4 策略管理器

如果需要在线更新自主构件的策略库,比如加入新的策略,则可以有如下两种方式:第 1 种是直接修改原有

的策略文件,Knowledge agent 监测到 Resource 的更新之后,会将策略文件重新编译并添加到知识库中;第 2 种是直接使用 Knowledge Builder 提供的接口,以控制台的方式编译策略文件,并添加到知识库中去.

3.2 基于质量动态评估的自适应

基于质量动态评估的自适应,重点是列出影响自主构件质量的所有因素,并动态维护和评估这些因素对构件质量的影响.在我们的构件模型中,信息的收集完全由功能模块的行为来实现,而影响质量的各个可控因素的取值和构件所选择的策略之间的映射则由容器负责同步.

图 5 是质量属性的示意图.其中,*subVar* 是由影响构件质量属性的所有因素(包括外部因素和内部因素)构成的元组.*subVar* 在构件实例化时被创建并初始化.*subVar* 的不同取值代表了构件的不同状态.*scoreTable* 是一个 HashMap,key 是 *N* 元组 *subVar*,value 是一个数值,用来记录 *subVar* 每个取值的得分.为了简单起见,我们的项目中将变量的类型统一设定为 int,score 的类型统一设定为 Double.基于质量动态评估的自适应过程是异步的,开发者需要定义一个采样间隔 *t*,每隔时间 *t*,质量管理器就会进行一个采集-分析-决策-执行的自适应循环,其中,采集过程主要是对影响质量属性的变量值和体现当前质量属性的信息进行收集、统计.分析过程主要是根据所收集的信息更新 *scoreTable*.决策过程是根据 *scoreTable* 和目前的环境状态决定如何进行自适应.执行过程则包含了自适应过程中的操作细节.另外,收集过程要收集的信息可能包含某种事件发生时的瞬时信息,因此也可以通过编写对构件容器的截取器事件的响应进行同步的信息采集工作(如图中的 *beforeExecution* 和 *afterExecution*).

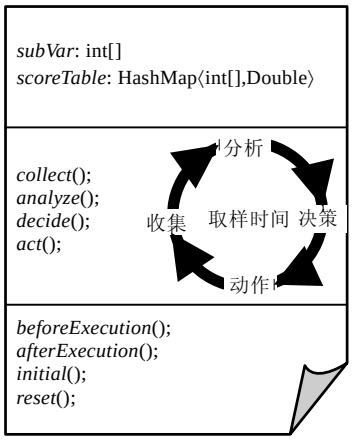


Fig.5 Quality

图 5 质量属性示意图

我们的自主构件是在 EJB 构件模型上进行扩展,加入了一些新的特征.一个功能模块就是一个普通构件(EJB).要把一个普通构件变成自主构件有两种方式,分别是:1) 配置文件法.这种方式不需要改变原来的普通构件,只需要编写配置文件,并将普通构件、策略文件、质量属性定义一起打包,部署运行即可;2) 标注法.这种方式借鉴了 EJB3.0 中 POJO 的思想,不需要编写配置文件.只要在普通构件中适当的地方加入标注即可:

```
@Strategy(strategyName="LightChange",qualityName="Reliability")
public LightState tick() {
}
```

这段代码的含义是:方法 tick()的实现是由名字为 LightChange 的策略来描述,并且该策略的不同实现将影响系统的 Reliability 质量属性,系统在运行时刻将会在名字为 LightChange 的实现策略中选择使得系统的 Reliability 质量属性较高的那个作为 tick()方法的实现体.图 6 是自主构件部署和运行时刻示意图.

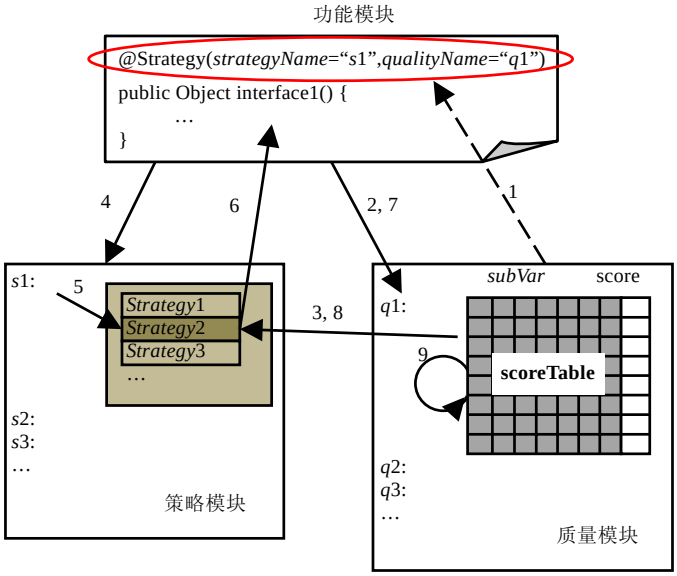


Fig.6 Deploy and running of autonomic component

图 6 自主构件的部署和运行

部署阶段,容器通过解析标注获取相应信息.在运行时刻:

步骤 1. 当实例化一个自主构件时,容器同时根据标注信息为该实例创建两个数据结构,即实现策略映射表和质量属性得分表(scoreTable);

步骤 2、步骤 7. 这两步是可选的,表示质量模块截获到接口被调用事件,在接口功能执行前/后对外部信息进行收集、统计、分析,并更新当前 subVar 值在 scoreTable 中的得分;

步骤 3、步骤 8. 这两步与步骤 2、步骤 7 相对应,表示质量模块根据更新后的 scoreTable 做出自适应调整,并将调整后的 subVar 值同步到策略模块中;

步骤 4. 策略管理器截获到对接口 interface1 的调用请求;

步骤 5. 策略管理器从实现策略映射表查到当前策略 *s*1 的实现策略是 *Strategy*2,并创建一个 *Strategy*2 的执行实例;

步骤 6. 策略管理器将执行结果返回给功能模块;

步骤 9. 表示质量属性 *q*1 在整个生命周期中始终异步地进行信息采集-分析-决策-执行过程,更新 scoreTable,并进行自调整.

4 实验及数据分析

在一些城市,拥堵的交通状况给人们的生活带来很多不便,而交通灯的策略对道路系统有着非常重要的影响.但是由于交通系统的复杂性,我们不可能制定出一套最优的交通灯控制策略来最大化司机的总体驾驶收益,这就需要系统在运行时刻动态学习如何根据不同的道路状况做出适应性的调整.我们将在下面的实验中展示如何用我们的自主构件模型实现交通灯控制系统,并用此系统进行了两个实验:第 1 个实验演示了由于不同的质量属性定义导致的自适应结果的差别,第 2 个实验演示了新策略的在线加载及其对系统产生的影响.

4.1 实验系统描述

我们设想了一个简化的道路系统,并在上面部署了交通灯自主构件来控制交通灯的转换,以达到降低交通拥堵程度的目的.假想的道路系统如图 7 所示,其中,空心三角代表车辆的行驶方向,横向和纵向道路的交点为十

字路口,由交通灯自主构件控制交通灯的转换.道路与边界有 12 个接点(每边 3 个),车辆将从这些接点进入/离开道路系统.由于现实世界中,上下班时道路比较拥堵,而其他时候相对比较清闲,因此我们设定系统开始的 3 000 个 tick(tick 为系统的时间单位)各个路口的发车率较低(每个入口大约每 100 个 tick 发 1 辆车),之后的 4 000 个 tick 为上班高峰期,左右两侧的路口发车率较高(每个入口大约每 50 个 tick 发 1 辆车),随后的 3 000 个 tick,各个路口的发车率恢复初始水平,再之后的 4 000 个 tick 为下班高峰期,上下侧路口的发车率较高.以上 4 个阶段为 1 个周期,系统不断重复这个周期.我们将针对该道路系统设计交通灯自主构件,以到达降低道路拥堵程度的目的.

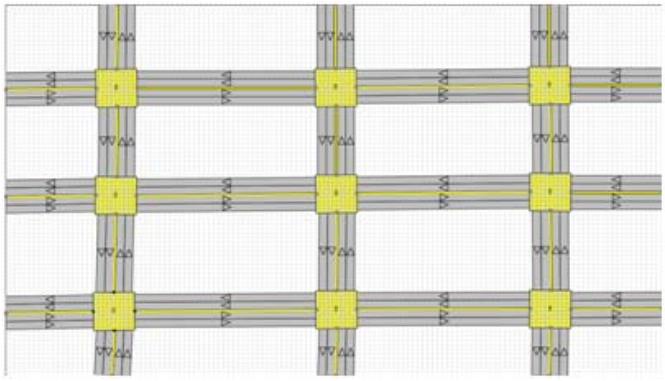


Fig.7 A simulated traffic system

图 7 假想道路系统

4.2 系统设计与实现

系统主要由道路系统构件和交通灯控制自主构件组成.其中,道路系统构件是普通构件,维护整个道路系统的所有信息,并提供一个图形界面.其每个 tick 更新系统中的车辆信息,并向交通灯自主构件查询交通灯状态,据此更新交通灯的颜色.道路系统对外提供路况信息查询的接口.交通灯控制自主构件会使用道路系统构件的路况查询服务,同时也为道路系统构件提供交通灯控制服务.每当道路系统构件调用交通灯控制自主构件时,交通灯控制自主构件就将自身的 tick 计数器加 1,同时将交通灯的更新状态返回给道路系统构件.下面我们主要关注交通灯自主构件的设计与实现.

首先是功能模块的设计,这部分的主要任务是设计好构件的基本数据结构及原子动作,并对外暴露其接口.例如,其基本数据包括:红绿灯转换间隔、黄灯持续时间、道路系统构件的引用、tick 计数器、各个红绿灯的状态等.原子动作包括将绿灯转为黄灯、将黄灯转为红灯以及各种向道路系统查询路况的动作等.最终,构件对外提供一个查询交通灯更新状态的接口,每个交通灯自主构件的实例只对一个道路系统构件的实例提供服务.

其次是策略的设计,这里的策略指的是红绿灯控制的策略,当道路系统调用构件的红绿灯控制服务时,构件将选择一种红绿灯控制策略来执行.不同的策略将导致不同的红绿灯转换方式,由于我们不能确定哪种策略比较有优势,所以开始的时候,先制定两种比较简单的策略:一种是较短的固定转换间隔的红绿灯转换策略,另一种是较长的固定转换间隔的红绿灯转换策略.如果在运行过程中发现更好的策略,则可以离线编写并编译,然后在线加入到系统的策略库中.

最后是质量模块的设计,如第 2.2.1 节所述,这一模块的主要任务是找出影响道路拥堵程度的各个因素,并给出一种对这些因素的不同取值进行评估的方案及根据评估结果进行自适应决策的方案.

1) 影响道路拥堵程度的因素

很显然,主要包括道路当前的拥堵程度、道路未来的发车状况、交通灯转换的策略.由于道路未来的发车状况会某种程度地反映到未来的道路拥堵程度中去,同时,由于我们所设想的场景的发车状况带有明显的周期性,因此为了提高自适应的效率,我们决定忽略掉这个因素.下面我们对道路拥堵程度和交通灯转换策略这两个

因素进行一个明确的定义.

道路拥堵率:道路系统中的车辆数目/道路系统所能容纳的最大车辆数(0%~100%).

考虑到自适应的效率问题,我们不可能对拥堵率的 0%~100%的 101 个取值都进行评估,因此规定:当拥堵率在 0%~20%时为清闲,对应的拥堵程度值为 1;当拥堵率在 20%~40%时为平均,对应的拥堵程度值为 2;当拥堵率在 40%以上时为繁忙,对应的拥堵程度值为 3.

交通灯转换策略:交通灯控制自主构件在决定交通灯状态时所采用的策略.

目前,我们的策略库中包含两种交通灯转换策略(即前面所说的较短固定转换间隔策略和较长固定转换间隔策略),按照在实现策略表中注册的顺序,值分别为 1,2,3,…

2) 动态自评估的方案

如何评估在特定的道路拥堵程度下哪种交通灯转换策略更优,不是一件很容易的事情.例如,在 30%的拥堵率下选择了交通灯转换策略 a,一段时间后拥堵率变为 20%;而在 25%的拥堵率下选择了交通灯转换策略 b,一段时间后拥堵率变为 18%.我们无法判断 a 策略更优还是 b 策略更优.因此,必须找到一种更合理的评估方案.在这里,我们给出一个主观的假设,就是在任意一种交通拥堵程度下,如果某种策略能够使得在其被选择之后的一段时间里,车辆从进入道路系统到离开道路系统所需要的平均时间更少,就说明该策略更优.然而,一段时间内,车辆通过道路系统的平均时间仍然较难测量,因此我们从另一个角度来估算这个值.假定每个 tick,道路系统中的所有车辆都付出一个单位的时间开销,那么在一段时间内道路系统中车辆的总时间开销就等于每个 tick 道路系统中的车辆数(*carsOnScreen*)的加总,这个值很容易获得.同时,在总开销相同的情况下,通过道路系统的车辆数越多,则平均时间就越短,反之则越长.而通过道路系统的车辆数可以参考该时间段内进入道路系统的车辆数(*carsEntered*)和离开道路系统的车辆数(*carsExited*),这两个值也容易获得.因此,我们就用一段时间内车辆总时间开销除以进入道路系统的车辆数(*averageTime*1)与车辆总时间开销除以离开道路系统的车辆数(*averageTime*2),这两个值的调和平均数来评估策略的好坏.

$$score = 1000 \times \left(\frac{1}{averageTime1} + \frac{1}{averageTime2} \right)$$
$$= 1000 \times \left(\frac{carsEntered}{totalDriveTime} + \frac{carsExited}{totalDriveTime} \right)$$
$$= 1000 \times \frac{carsEntered + carsExited}{\sum carsOnScreen},$$

其中,1 000 是一个修正因子,它使得 *score* 的值更便于比较.

3) 自适应决策方案

我们设计的这个过程比较简单,有如下 3 条基本原则:

第一、当有新的策略加入时,优先试用该策略;

第二、当一种策略被选中时,由于需要过一段时间才能观测到它对系统产生的影响,因此规定在该策略被评估之前赋予其最高的优先级;

第三、除了以上两种情况之外,任何时刻都选择在当前拥堵程度下,评估得分最高的那种策略.

4) 在设计完成之后,功能模块、策略、质量这 3 部分通过标注的方式联系到一起

```
public class TrafficLight implements TrafficLightRemote, AutonomousComponent {
    @DependencyService (serviceName="RoadNetwork")
    private RoadNetworkRemote roadNetwork;
    @Strategy (strategyName="LightChange",qualityName="Reliability")
    public LightState tick() {
    }
}
```

4.3 实验结果与分析

我们分别进行了两个实验.第 1 个实验观察了在不同的质量属性定义下,系统的自适应结果.这两种质量属性定义在质量影响因素的选择、动态评估方案和决策方案的设计上基本没有区别,唯一的区别在于第 1 种质量定义在策略被选择后的第 750 个 tick 进行评估,第 2 种质量定义在策略被选择后的第 3 000 个 tick 进行评估.图 8 是系统的运行情况,横坐标代表系统运行时间,纵坐标代表道路拥堵率.

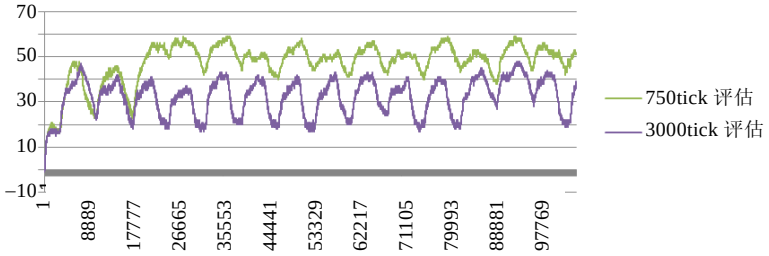


Fig.8 Result of experiment 1

图 8 实验 1 的运行结果

从图中可以明显地看出,从大概 17 000tick 之后,以 750tick 为评估时机的道路拥堵率明显高于以 3 000tick 为评估时机的道路拥堵率.我们让系统输出了策略的选择情况,发现 17 000tick 之后,在以 750 个 tick 为评估时机的实验中,系统大部分时间选择了第 1 种策略(固定转换间隔为 50),而在以 3 000 个 tick 为评估时机的实验中,系统大部分时间选择了第 2 种策略(固定转换间隔为 150).这说明,系统在 17 000 个 tick 的试用期之后进入了稳定状态,这个自适应的效率还是不错的.同时,无论选用哪种交通灯转换策略,系统的拥堵率都在 20%以上.这说明,第 2 种策略在平均和繁忙的拥堵程度下是严格优于第 1 种策略的.之所以两个实验会产生截然不同的结果,是因为 750 个 tick 的评估时机太短,新选中策略还不能发挥其影响,导致第 2 种策略来不及改善第 1 种策略造成的高拥堵率,从而得不到较高的评分.而前 17 000 个 tick 两个实验的拥堵率比较相近,是因为这个阶段属于试用阶段,所以二者相差不大.

第 2 个实验展示了在发现已有策略的不足之后,在线加入新的策略,系统的运行情况.这个实验中采用的质量评估时机为 3 000tick.由于系统有上班高峰期(左右两侧发车率升高)和下班高峰期(上下两侧发车率升高),所以我们认为,固定的交通灯转换间隔可能不合理,而且在对系统模拟运行进行观察的过程中也发现交通灯绿灯侧比较清闲,而红灯侧排起长队的情况,因此我们设计了第 3 种策略:当红灯侧 150 个像素内的车辆总数大于绿灯侧 150 个像素内的车辆总数时就转换交通灯.同时,为了防止交通灯频繁转换,我们规定转换的最少间隔为 80 个 tick.将写好的策略文件编译并在第 60 000 个 tick 左右时加入到系统规划库中,运行结果如图 9 所示.

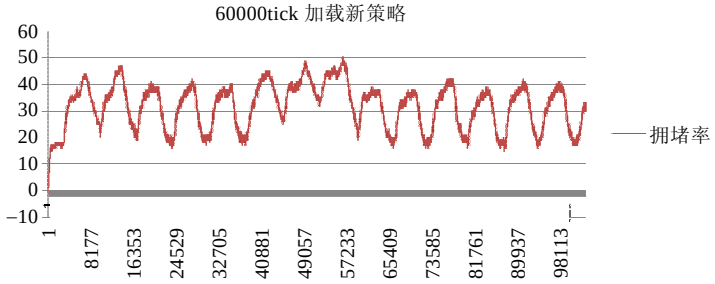


Fig.9 Result of experiment 2

图 9 实验 2 的运行结果

从图中可以看出,在 60 000 个 tick 之前,系统拥堵率在高峰期经常达到 40%以上,有时甚至达到近 50%;而在 60 000 个 tick 之后,系统拥堵率则很少在 40%以上.从系统输出的策略选择上看,在 6 000 个 tick 之后,系统在平均和繁忙状态下大部分选用策略 3,偶尔选择策略 2.而在清闲状态下,主要使用策略 2.这从直觉上也容易理解,在清闲状态下,不同策略的差别应该不大(红绿灯转换间隔特别长的策略除外).

5 相关工作

对于自主实体进行建模的问题,目前主要的实现手段是将 Agent 领域在决策技术研究方面的成果与当前主流的软件开发技术相结合,以描述现有计算资源的自主行为.近年来,不少研究项目对支持自适应行为的构件模型进行了讨论.文献[5]介绍了一种对构件模型 Fractal 进行扩展以支持自适应行为的方法,其所做的扩展主要分为 3 个部分:提供一个收集环境上下文信息的服务以获取运行时刻信息、基于“事件-条件-动作”的模式定义自适应策略以及使用一个元层对象协议来截取和执行计算功能.Accord[6]是一个基于高层规则进行组织和协同的构件模型,其特点在于将构件的计算行为与构件的组装行为相分离,且上述计算行为、组装行为均通过一组高层规则来管理.文献[7]提出了一种构件框架 *K*-component,支持构件通过分布式强化学习技术在运行时刻动态调整自身的策略.这些构件模型的一个共同特点是:基于规则的自适应,即构件根据瞬时的环境信息或事件的触发进行自适应调整,而很多复杂系统的自适应无法或者很难用规则来表示.除此之外,Fractal 和 *K*-component 都不支持策略的在线更新.

6 总结及展望

基于 Internet 的软件系统已成为一种新的软件形态,开发这样的系统面临着很多亟待解决的问题和挑战.本文描述了一种基于质量动态评估的自主构件原理及实现,试图通过对实现自主构件的方法和手段的探索,为开发具有自主性的基于 Internet 的软件系统提供一定的实践基础和经验.

在实现自主构件时,我们采用了扩展已有的 EJB 构件模型的方式,向 EJB 中加入了具有自主构件特征的策略模块和质量模块,并通过扩展标准的 EJB 容器,用标注的方式将这 3 个部分联系起来.策略模块通过集成已有的 Drools 规则引擎实现,支持策略的动态定制,能够在不修改功能模块的情况下改变构件的自适应策略甚至服务实现策略.质量模块通过向容器中加入质量管理器来实现.质量管理器提供一套动态质量评估 API,并负责维护质量评估表与构件自适应行为之间的映射.由于复用已有的构件和基础设施,这也使得本文的方法具有更高的可行性和更大的应用范围.

但目前阶段,我们所研究的只是自主计算实体的建模与实现.自主构件必须复用到软件系统中,参与系统的行为,并与其他自主构件合作才能实现系统的目标.为了保证自主构件之间的交互能够最终实现系统的目标,自主构件之间的交互必须根据系统的需要而定.但自主构件的开发和实现是独立的,它们可能并不知道将要复用到什么系统中去,也不知道系统要求自主构件如何交互.因此,下一阶段我们将进一步研究如何运用自主构件来构造软件系统.

致谢 在此,我们要衷心感谢北京大学软件工程实验室 pkuas 小组的所有老师和同学,他们在系统实现过程中提出过很多宝贵的指导性意见和建议,感谢周金果、马剑竹等同学,他们都参与讨论了系统设计,也参加了部分代码的编写工作,其中自主构件的定制工具由周金果同学协助完成.

References:

[1] Karsai G, Sztipanovits J. A model-based approach to self-adaptive software. IEEE Intelligent Systems, 1999,14(3):46−53. [doi: 10.1109/5254.769884]

[2] Kephart JO. Research challenges of autonomic computing. In: Proc. of the 27th Int'l Conf. on Software Engineering. 2005. 15−22. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1553533 [doi: 10.1145/1062455.1062464]

[3] Patil RS, Fikes RE, Patel-Schneider PF, Mckay D, Finin T, Gruber T, Neches R. The DARPA knowledge sharing effort: Progress report. In: Proc. of the Knowledge Representation and Reasoning (KR&R'92). 1992. 777−788. http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.13.1180

[4] Kephart JO, Chess DM. The vision of autonomic computing. IEEE Computer, 2003,36(1):41−50. [doi: 10.1109/MC.2003.1160055]

[5] David PC, Ledoux T. Towards a framework for self-adaptive component-based applications. In: Proc. of the Int'l Conf. on Distributed Applications and Interoperable Systems. 2003. 1−14. http://www.springerlink.com/content/fu232qa7wl6lrr2e/ [doi: 10.1007/978-3-540-40010-3_1]

[6] Liu H, Parashar M, Hariri S. A component-based programming model for autonomic applications. In: Proc. of the Int'l Conf. on Autonomic Computing. 2004. 10−17. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1301341 [doi: 10.1109/ICAC.2004.2]

[7] Dowling J. The decentralised coordination of self-adaptive components for autonomic distributed systems [PH.D. Thesis]. Dublin: University of Dublin, 2004.

[8] Laddaga R. Creating robust software through self-adaptation. IEEE Intelligent Systems, 1999,14(3):26−29. [doi: 10.1109/MIS.1999.769879]

[9] Cheng BHC, de Lemos R, Giese H, Inverardi P, Magee J, Andersson J, Becker B, Bencomo N, Brun Y, Cukic B, Serugendo GM, Dustdar S, Finkelstein A, Gacek C, Geihs K, Grassi V, Karsai G, Kienle HM, Kramer J, Litoiu M, Malek S, Mirandola R, Müller HA, Park S, Shaw M, Tichy M, Tivoli M, Weyns D, Whittle J. Software Engineering for Self-Adaptive Systems: A Research Roadmap. Leibniz: Schloss Dagstuhl, 2008. 1−26. [doi: 10.1007/978-3-642-02161-9_1]

[10] Salfner F, Lenk M, Malek M. A survey of online failure prediction methods. ACM Computing Surveys, 2010,42(3):1−68. [doi: 10.1145/1670679.1670680]

[11] Fang X, Ma JZ, Wang MG, Jiao WP. An approach to the automated integration of multi-agent systems based on two-way selections. Computer Engineering and Science, 2010,32(6):68−73 (in Chinese with English abstract).

[12] Drools. 2008. http://labs.jboss.com/portal/jbossrules/

[13] Sun X, Zhuang L, Liu W, Jiao WP, Mei H. A customizable running support framework for autonomous components. Journal of Software, 2008,19(6):1340−1349 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/19/1340.htm [doi: 10.3724/SP.J.1001.2008.01340]

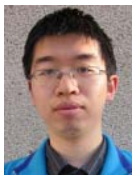
附中文参考文献:

[11] 方汐,马剑竹,王茂光,焦文品.一种基于双向选择的多 agent 系统自动集成方案.计算机工程与科学,2010,32(6):68−73.

[13] 孙熙,庄磊,刘文,焦文品,梅宏.一种可定制的自主构件运行支撑框架.软件学报,2008,19(6):1340−1349. http://www.jos.org.cn/1000- 9825/19/1340.htm [doi: 10.3724/SP.J.1001.2008.01340]

接钧靖(1986－),男,山东烟台人,博士生,主要研究领域为复杂系统自组织,自主构件技术.

焦文品(1969－),男,博士,副教授,CCF 高级会员,主要研究领域为软件复用,智能软件.



史庭训(1988－),男,博士生,主要研究领域为自主构件技术,物联网相关技术.

孟繁晶(1976－),女,博士,高级研究员,主要研究领域为资产收集与复用,软件开发治理,云转型技术.