软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2012,23(11):2905−2922 [doi: 10.3724/SP.J.1001.2012.04301]

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

多 Agent 系统的上下文感知增强*

马 骏[1,2], 陶先平[1,2+], 朱怀宏[1,2], 吕 建[1,2]

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

[2](南京大学 计算机软件研究所,江苏 南京 210093)

Enhancing Multi-Agent System with Context-Awareness

MA Jun[1,2], TAO Xian-Ping[1,2+], ZHU Huai-Hong[1,2], LÜ Jian[1,2]

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

[2](Institute of Computer Software, Nanjing University, Nanjing 210093, China)

+ Corresponding author: E-mail: txp@nju.edu.cn

Ma J, Tao XP, Zhu HH, Lü J. Enhancing multi-agent system with context-awareness. *Journal of Software*, 2012,23(11):2905−2922 (in Chinese). http://www.jos.org.cn/1000-9825/4301.htm

Abstract: Multi-Agent system (MAS) is widely used to develop applications in different domains. Currently, the computing platform is becoming more and more open, dynamic, and uncontrollable. Hence, software systems are required to adapt to the changing states of themselves and their running environments. In other words, systems should be context-aware. However, the way to enhance existing MAS applications with context-awareness is not well addressed by existing works. In this paper, based on the Separation of Concerns principle, the study combines context-oriented programming (COP), reflection, as well as code instrumentation technologies in a way to introduce context-awareness to existing MAS applications. With the proposed approach, agents of an existing MAS application are transformed to context-aware ones, even if the source code is unavailable. In addition, with the help of the underlying runtime environment, the administrator can dynamically adjust the context-aware behaviors of a specified agent (or a group of agents) at runtime.

Key words: software agent; multi-agent system; context-aware; programming; software methodology

摘 要: 如今,多 agent 系统(multi-agent system,简称 MAS)被广泛用于开发各种应用系统.当前,开放、动态、难控的计算平台要求软件系统能够根据系统自身及其环境状态信息及其改变,动态地调节自身的行为,即具备一定的上下文感知能力.然而,现有工作并未就如何向既有的 MAS 应用系统引入上下文感知能力提出有效的解决方案.依照关注分离原则,结合面向上下文程序设计技术(context-oriented programming,简称 COP)、反射技术(reflection)以及代码植入技术(code instrumentation),提出了一套 MAS 系统上下文感知增强框架和底层支撑技术.开发人员可以在既有应用源码不可得的情况下,自动地将指定 agent 类型转换为(扩展为)具有上下文感知能力的 agent 类型.此外,利用底层运行支撑环境,系统管理员可以在系统运行时刻动态地调整指定 agent 的上下文感知行为.

关键词: 软件 agent;多 agent 系统;上下文感知;程序设计;软件方法学

* 基金项目: 国家自然科学基金(61073031, 61021062); 国家重点基础研究发展计划(973)(2009CB320702); 国家高技术研究发展计划(863)(2012AA011205)

收稿时间: 2012-06-08; 定稿时间: 2012-08-21

中图法分类号: TP18 文献标识码: A

软件 agent 与生俱有的自主性、智能性、移动性、协同性等特征,使其尤为适合分布式、智能化应用系统[1],为分布式开放系统的分析、设计和实现提供了一条行之有效的途径.目前,基于多 agent 系统(multi-agent system,简称 MAS)的软件系统设计和开发技术得到了学术界和企业界的广泛关注,MAS 系统已被应用于众多应用领域[2−6].随着 Internet 的广泛普及、无线通信技术的迅速发展以及新型计算设备的层出不穷,计算机软件系统所面临的计算平台和环境经历了一个由早期的封闭、静态、易控到如今的开放、动态、难控的转变历程[7−9],而且这一转变还在继续,并且更加迅猛.开放、动态、难控的计算环境使得对软件开发、部署、运行和维护的外部环境在通常只能做较少的界定或难以明确界定,从而导致构成外部环境的要素及其关系在软件生存周期中是动态可变的,构成外部空间的各个要素及其关系的内容的变化方式常常是难以控制的[8].面对开放、动态、难控的计算环境,出现了众多新兴的计算模型和模式(例如,网构软件模型[7−9]、普适计算[10,11]等);为了适应环境开放、动态、难控的特点,这些新兴模型或模式往往都要求软件系统能够根据自身和所处环境的状态及其变化来动态调节自身的行为,即具备一定的上下文感知能力[12,13]或自适应能力[14].

为了适应开放、动态、难控的计算平台,国内外不少研究人员利用 MAS 设计和开发具有上下文感知能力的自适应软件系统.具有上下文感知能力的 MAS 系统是目前国内外的研究热点[15].一方面,MAS 被直接用于开发具有上下文感知能力的应用软件系统[5,6,16];另一方面,MAS 技术也被广泛用于实现一些上下文感知应用的支撑平台或框架[9,17−25],为上层上下文感知自适应应用系统的开发、运行等各方面提供统一的、不同程度的支持.

现有的这些框架和平台,确实为从无到有开发一个全新的上下文感知 MAS 应用提供了一定的支持.然而,由于 MAS 已被应用到众多领域[2−6],现阶段存在相当数量的非上下文感知的 MAS 应用系统.对于使用这些既有 MAS 系统的用户而言,往往并不希望放弃现有的系统而重新设计和开发一套业务功能类似的、全新的上下文感知应用;因为这样意味着需要根据所使用的底层框架或平台的要求来重新设计和实现目标系统,必然导致大量的时间、人力和经费的开销.相对而言,这些用户更希望能够尽可能地复用现有的系统——在现有 MAS 系统的基础上稍作修改就能添加一定的上下文感知功能以满足新的需求,即对现有 MAS 系统进行上下文感知增强.而现有框架并未就此提供一个有效的解决方案.此外,在设计和开发上下文感知应用系统时,研发人员往往不可能预测到系统在最终部署、运行时所会遇到的各种情况,因此需要能够在系统运行的过程中,动态地添加、删除和修改系统的上下文感知行为.

为此,本文提出了一套 MAS 系统上下文感知增强技术和方法框架.特别地,该框架通过对既有 MAS 应用系统的进行一次离线的自动转换来引入上下文感知能力;对于转换得到的应用系统,通过独立定义一系列的情境、操作变体以及上下文感知策略,并利用底层运行支撑(动态的加载或卸载上述相关定义),可以在运行时刻在线调整应用的上下文感知行为.简单来说,本文主要贡献包括以下几点:

- 提出一个抽象上下文感知 agent 模型,为上下文感知 MAS 应用系统的设计和开发提供一个参考.
- 基于上述上下文感知 agent 模型,结合面向上下文编程(context-oriented programming,简称 COP)[26]技术、反射技术(reflection)以及代码植入技术(code instrumentation),提出了一套 MAS 系统上下文感知增强的方法框架.该框架充分体现关注分离(separation of concerns)原则,开发人员可以独立地开发应用的主体业务逻辑与上下文感知逻辑.利用加载时刻代码植入技术,支持在既有 MAS 应用系统的源码不可获取的情况下引入上下文感知能力.此外,经过转换的系统在运行时刻可以动态地添加、删除或者替换应用的上下文感知行为.
- 根据上下文感知 agent 模型,本文还给出了一个原型系统的实现,并在该系统上实现一个简单应用来展示本文所提出的技术方案的可行性.

本文第 1 节给出一个简单 MAS 案例.第 2 节着重介绍上下文感知 agent 模型.第 3 节详细介绍利用上下文感知 agent 模型进行 MAS 应用系统上下文感知增强的方法框架以及底层运行支持环境的基本构成.第 4 节给出一个原型系统以及基于该系统的一个案例实现.第 5 节简单介绍与本文相关的现有工作.第 6 节总结本文工

作并讨论我们今后的工作.

1 一个简单的电子商务 MAS 系统案例

首先,我们来考虑一个简化的电子商务 MAS 系统:整个系统仅包含 3 类 Agent:*SellerAgent*,*BuyerAgent* 以及 *MarketAgent*.其中,*SellerAgent* 代表卖家出售商品(为了简化,这里假定每个商品都由一个 *itemID* 唯一确定),提供以下基本的操作:*getPrice*(int *itemID*)以及 *orderItem*(int *itemID*,int *clientID*).其中,*getPrice*(int *itemID*)获取 *itemID* 所指定商品的价格,*orderItem*(int *itemID*,int *clientID*)则接受来自买家(由 *clientID* 所指定)的订购商品(由 *itemID* 所指定)的请求;*BuyerAgent* 则代表具体的买家自动订购指定的一系列商品;*MarketAgent* 表示一个卖场,它提供一个基本的交易平台,统一管理所有的 *SellerAgent* 和 *BuyerAgent*.*MarketAgent* 提供给 *BuyerAgent* 一个查询接口 *lookupSeller*(int *itemID*),用于查询出售 *itemID* 所指定商品的所有卖家的列表.

应用的基本流程简单描述如下:卖家创建 *SellerAgent* 实例,向 *MarketAgent* 注册新创建的实例以及所售商品列表;买家创建 *BuyerAgent* 实例,指明所需订购商品的列表;*BuyerAgent* 随后通过 *MarketAgent* 的 *lookupSeller*(int *itemID*)接口查询出售所需商品的卖家的列表;获取列表后,*BuyerAgent* 进一步向列表中的各个买家查询所需商品的价格,并返回给用户一个最优订购计划(此处,针对每个所需购买的商品,最优订购计划记录了出售该商品的卖家中售价最低的卖家信息);最终,*BuyerAgent* 执行最优订购计划,通过 *SellerAgent* 的 *orderItem*(int *itemID*,int *clientID*)接口向最优购买计划中不同 *SellerAgent* 订购指定商品.

假定为了促销,卖场负责人提出一个新的需求:(*R*-1)要求所有卖家在“五一”假期以及“国庆”长假等节假日期间都对所售商品统一打折(比如九折)销售.实现这一新的需求可以有多种方式:

1. 各卖家人为地修改商品的价格或其他配置文件;
2. 重新编写或继承 *SellerAgent* 类(重写或重载 *getPrice*());
3. 利用 AspectJ(http://www.eclipse.org/aspectj/)对 *SellerAgent* 代码进行修改(如图 1 所示);
4. 基于现有上下文感知框架和平台重新编写应用.

```
1. pointcut getPrice(): call (double SellerAgent.getPrice(int));
2. double around(): getPrice(){
3.     Data now=new Data();
4.     if (isHoliday(now)){
5.         return proceed()*0.9;
6.     } else {
7.         return proceed();
8.     }
9. }
```

Fig.1 Introducing context-awareness via AspectJ

图 1 利用 AspectJ 引入上下文感知

方案 1 不需要对现有应用系统做任何修改,但是给卖家带来了大量的工作负担,而且容易出错.应用系统所面临的计算平台具有开放、动态、难控的特征,系统研发人员难以在系统设计和开发之初就确定系统所需的上下文感知行为,往往需要在系统运行过程中动态的添加(例如,为了进一步促销,卖场负责人又要求(*R*-2)所有卖家在每个周末都对所售商品统一打折(比如九五折)销售);而方案 2 和方案 3 都需要对系统离线地(off-line)重新编译并替换现有的文件,因此它们并不能较好地适应于需求动态变化频繁的情况(特别地,如果使用继承方式来实现,频繁的更新还会使程序的类层次结构越来越复杂,不利于系统进一步的维护和演化).尽管利用在线(on-line)更新技术,我们可以对第 2 种和第 3 种方式进行扩展,使其达到在线动态引入新的上下文感知需求功能的目的;但这两种方式根本上说并没有提供一套良好的框架或机制来引入上下文以及上下文感知逻辑,其上下文感知逻辑仍旧与应用逻辑交织在一起(例如,图 1 中用于处理上下文感知行为的 if 语句),耦合度比较高,并不适应上下文感知逻辑比较复杂的情况.为了进一步解耦感知逻辑与应用逻辑,我们可以利用第 4 种方案,即用现有的一些成熟上下文感知框架和平台对整个应用进行重新设计和开发.但是,这种方式的复用率较低、开销巨

大,仅适合于小规模应用系统的转换.

针对上述方案的不足,本文提出一个上下文感知增强Agent模型,并在此模型上,为既有MAS应用系统引入上下文感知能力提供了一套通用的、灵活的解决方案.通过该方案,用户可以在既有MAS应用源码不可获取的情况下对系统中的agent进行离线自动转换,使其具备上下文感知的基本能力.特别地,该方案遵循关注分离原则,上下文感知逻辑与应用逻辑独立开发,系统管理员能够在运行过程中对转换后的应用系统在线动态地添加、删除上下文感知行为.

2 上下文感知增强 Agent 模型

2.1 上下文感知Agent

通常来说,一个MAS应用系统往往是由1个或多个不同功能的agent所构成.其中,每个agent维护一定的数据结构并提供对这些数据的一系列的操作(operation);agent之间通过消息传递来沟通、共同协作完成既定的应用目标.抽象来看,一个agent可以简单表示为$\langle id,\mathbb{D},\mathbb{O},\mathbb{I}\rangle$(此处我们不考虑agent的BDI模型).其中,$id$表示agent的唯一的身份标识;$\mathbb{D}$则表示agent所维护的数据;$\mathbb{O}$表示agent所能对$\mathbb{D}$中数据进行的操作的集合;$\mathbb{I}$则表示agent与外界的通信接口(interface)的集合,用于刻画agent与外界(其他agent)的联系.而agent在运行时刻的行为则体现为一系列的操作以及通信(接受外界消息以及向其他agent发送消息)所构成的动态序列.

而上下文感知应用要求agent应该能够主动地调节自身,以适应其自身、外部运行环境的动态变化.抽象来看,一个上下文感知agent可以在现有$\langle id,\mathbb{D},\mathbb{O},\mathbb{I}\rangle$抽象结构上扩展了$\mathbb{S},\mathbb{V}$以及$\mathbb{P}$而获得,即可表示为$\langle id,\mathbb{D},\mathbb{O},\mathbb{I},\mathbb{S},\mathbb{V},\mathbb{P}\rangle$.其中,$\mathbb{S}$表示agent所关注的环境状态,即其情境信息(简称情境(situation))的集合,$\mathbb{V}$表示该agent的操作变体(operation variant)的集合,而$\mathbb{P}$表示agent所加载的上下文感知策略(context-aware policy)的集合.$\mathbb{S},\mathbb{V},\mathbb{P}$三者共同刻画了agent的上下文感知能力.

- 情境集合$\mathbb{S}$

一个系统的环境往往是由众多的环境要素所构成,每一个环境要素刻画了系统某一个特定的属性(attribute).我们用$\mathbb{A}$表示系统中所有环境要素所构成的集合,并且针对一个具体环境要素$a\in\mathbb{A}$,用$Dom(a)$表示其取值范围.

在某一时刻,$\mathbb{A}$中所有环境要素及其取值($value$)所构成的属性-值对的集合为系统在这一时刻的一个快照(snapshot),它刻画了系统在这一特定时刻所处的状态.一个实体快照可以表示为

$$s \doteq \{\langle a,v\rangle \mid a\in\mathbb{A}, v\in Dom(a)\} \tag{1}$$

其中,s满足以下条件:

(1) $\mathbb{A}=\bigcup_{\langle a,v\rangle\in s} a$,即一个快照刻画了系统所有环境要素的取值状态;

(2) $\forall\langle a,v\rangle,\langle a',v\rangle\in s \Rightarrow a\neq a'$,即同一要素在同一时刻的取值具有唯一性.

在此基础上,我们用$\mathbb{G}$表示系统所有可能的快照的集合,并称其为快照空间(snapshot space).

一个情境(situation)定义为关于系统中某一个(或多个)环境要素的命题(我们用$factor(s)$表示情景s所涉及到的所有环境要素的集合),其真伪取决于当前的快照.给定一个上下文感知agent,其所关注的情境往往是有限的,而且不同agent所关注的情境可以是不同的.为此,针对某个agent(例如agt)我们用$\mathbb{S}(agt)$表示其所关注的所有情境构成的集合.假定一个MAS系统的所有agent所构成的集合为$\mathcal{A}$,我们可以进一步得到该系统所有agent所关注的情境的集合:

$$\mathbb{S}(\mathcal{A}) \doteq \bigcup_{agt\in\mathcal{A}} \mathbb{S}(agt) \tag{2}$$

在一指定时刻,agent需要根据当前快照s来决定$\mathbb{S}(agt)$中每一个具体情境的真伪,我们称这个过程为情境

评估(situation evaluation).情境评估过程可以表示为一个 $eval:\mathcal{G}\times\mathcal{S}(\mathcal{A})\to\{0,1\}$ 函数:

$$eval(\mathfrak{s},s)=\begin{cases}0, & \text{情境}s\text{在快照}\mathfrak{s}\text{中为假}\\ 1, & \text{情境}s\text{在快照}\mathfrak{s}\text{中为真}\end{cases} \tag{3}$$

给定一个情景集合$\mathcal{S}$以及一个快照 $\mathfrak{s}$,我们可以进一步将获取"$\mathcal{S}$中所有在 $\mathfrak{s}$ 中评估为真的情境所构成的集合"的过程表示为一个 $act:\mathcal{G}\times 2^{\mathcal{S}(\mathcal{A})}\to 2^{\mathcal{S}(\mathcal{A})}$ 的函数(此处 $2^{\mathcal{S}(\mathcal{A})}$ 表示 $\mathcal{S}(\mathcal{A})$ 的幂集):

$$act(\mathfrak{s},\mathcal{S})\doteq\{s\mid s\in\mathcal{S},eval(\mathfrak{s},s)=1\} \tag{4}$$

而一个 agent(例如 agt)感知环境状态的过程则体现为获取快照 $\mathfrak{s}$ (实际上,agt 仅需探知 $\mathfrak{s}$ 的一个子集 $\mathfrak{s}(\mathcal{S}(agt))$ 即可,$\mathfrak{s}(\mathcal{S}(agt))\doteq\{\langle a,v\rangle\mid\langle a,v\rangle\in\mathfrak{s}\wedge(a\in factor(s),s\in\mathcal{S}(agt))\}$)并根据快照进一步对$\mathcal{S}(agt)$中情境进行评估,最终获取 $act(\mathfrak{s},\mathcal{S}(agt))$ 的过程.

- 操作变体集合 $\mathbb{V}$

考虑到 agent 运行时刻的行为实际上由一系列基本操作的序列所构成,在我们的模型中将 agent 的操作视为 agent 运行时刻行为的最基本单位,并在此基础上引入 agent 的上下文感知能力.具体而言,针对 agent 的每一个操作 $o\in\mathcal{O}$,具有一组与之对应的操作变体(operation variant),用 $\mathbb{V}(o)$表示;这里的每一个操作变体 $ov\in\mathbb{V}(o)$ 与 COP[26]中的 *partial method* 类似,刻画了一个在特定情境条件下会激活的特殊操作.当 agent 执行操作 $o\in\mathcal{O}$时,将根据当前的情境信息 $act(\mathfrak{s},\mathcal{S}(agt))$ 以及$\mathbb{P}$中所定义的上下文感知策略决定是否执行以及如何执行其操作变体 $ov\in\mathbb{V}(o)$.

- 上下文感知策略集合$\mathbb{P}$

为了刻画 agent 的上下文感知行为,我们利用上下文感知策略来描述 agent 所关注的情境同操作变体之间的联系.一个上下文感知策略可以表示为 $cp\doteq\langle o,s,ov,t,p,l_b\rangle$.其中,$o\in\mathcal{O}$,表示 agent 的一个操作,这里称为基操作;$s\in\mathcal{S}$为一个情景,这里称为该策略的卫士情景(*guard situation*);$ov\in\mathbb{V}(o)$为操作 o 的一个操作变体;$t\in\{before,around,after\}$表示该策略所约束的操作变体 ov 在具体组合时候的编织类型(*weaving type*);p 是一个自然数,表示该策略的优先级;$l_b\in\mathcal{S}$表示该策略所屏蔽的情境构成的列表.在不考虑 t,p 以及 l_b 的情况下,策略 $cp\doteq\langle o,s,ov,t,p,l_b\rangle$ 的基本含义是指:"如果 agent 执行操作 o 的时候,卫士情境 s 被评估为真(即 $s\in act(\mathfrak{s},\mathcal{S})$),则操作变体 ov 应该被组装至此次调用所生成的操作链(参见第 2.2 节)之中并执行".我们用$\mathbb{P}(o)$来表示与某个给定的操作 o 相关的所有上下文策略所构成的集合($\forall cp\in\mathbb{P}(o),cp.o=o$).

类似于 AspectJ,这里同样引入 3 种不同的编织类型(*weaving type*).简单来说,$cp.t=before$ 表示当策略 cp 所约束的操作变体 $cp.ov$ 在被组装进最终的操作链时,应该放在基操作($cp.o$)之前;$cp.t=around$ 表示操作变体 $cp.ov$ 在被组装时应替换原有的 $cp.o$;而 $cp.t=after$ 表示操作变体 $cp.ov$ 在组装时应该放在 $cp.o$ 之后.

给定两个策略 $cp_1,cp_2\in\mathbb{P}(o)$,且 cp_1,cp_2 具有相同的编织类型($cp_1.t=cp_2.t$),若 cp_1 的优先级比 cp_2 高($cp_1.p>cp_2.p$),则当 $cp_1.s,cp_2.s$ 都为真时,最终组合得到的操作链中 $cp_1.ov$ 将先于 $cp_2.ov$;若两者优先级相同($cp_1.p=cp_2.p$),$cp_1.ov$ 与 $cp_2.ov$ 编入操作链的顺序可以是随机的,或者由用户指定一个顺序(例如,在我们的原型系统中,此时我们根据 cp_1,cp_2 加载时刻的先后来排序,先加载的策略所约束的操作变体排在前面).

通过刻画 $cp.l_b$,一个上下文感知策略 cp 可以屏蔽其他的策略.具体而言,策略 cp 屏蔽了一组策略 $\mathcal{B}(cp)$:

$$\mathcal{B}(cp)\doteq\{cp'\mid cp'\in\mathbb{P}(o),cp'.t=cp.t,cp.p>cp'.p,cp'.s\in cp.l_b\} \tag{5}$$

其中,任意一个策略 $cp'\in\mathcal{B}(cp)\subset\mathbb{P}(o)$ 都满足以下 3 个条件:(1) cp'具有和 cp 相同编织类型($cp.t=cp'.t$);(2) cp'的优先级比 cp 要低($cp.p>cp'.p$);(3) cp'的卫士情境被 cp 的屏蔽情境列表所包含($cp'.s\in cp.l_b$).在系统运行时刻执行操作 o 时,如果策略 cp 的卫士情境 $cp.s$ 评价为真,且 cp 不被其他优先级比他高的策略所屏蔽,所有 $\mathcal{B}(cp)$ 中的策略 cp'都将被其屏蔽,进而 $cp'.ov$ 将不会被组装到最终生成的操作链中.

2.2 上下文感知与操作链

当执行一个上下文感知 agent(例如 agt)的操作 $o\in\mathcal{O}$时,具体的执行效果取决于当前的情境($act(\mathcal{s},\mathcal{S}(agt))$)以及 agent 所加载的与操作 o 相关的上下文感知策略($\mathcal{P}(o)$).简单来说,agt 会根据当前情境创建一个操作链(*OperationChain*),然后进一步执行该操作链,得到最终的上下文感知执行效果.

如图 2 所示,对应于上下文感知策略的 3 种不同编织类型,一个操作链实际上是由 *BeforeChain*, *AroundChain*,*AfterChain* 这 3 个子链所构成的,每个子链都包含一组操作变体.顾名思义,*BeforeChain*, *AroundChain* 以及 *AfterChain* 子链中的各操作变体分别来自于 *agt* 所加载的编织类型为 *before*,*around* 和 *after* 的策略定义.

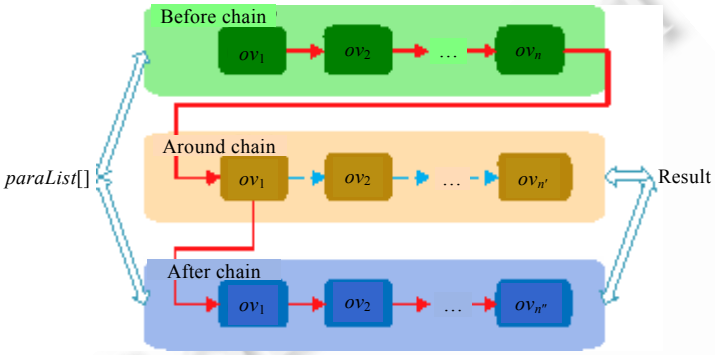

Fig.2 Construction and execution of an operation chain

图 2 操作链的创建与执行

具体而言,当执行操作 $o\in\mathcal{O}$时,*agt* 从$\mathcal{P}(o)$中依次逐个选取优先级最高的、没有被已选取策略屏蔽的、满足 $cp.t=before$ 且 $cp.s\in act(\mathcal{s},\mathcal{S}(agt))$ 的策略 cp,将 $cp.ov$ 插入到 *BeforeChain* 的尾部,从而得到此次执行 o 所对应的 *BeforeChain*.类似地,*agt* 可以创建针对此次执行 o 所对应的 *AroundChain* 以及 *AfterChain*.

当执行一个具体的操作链时,首先执行 *Before-Chain*,然后是 *AroundChain*,最后执行 *AfterChain*.针对一个具体的子链(*BeforeChain* 或 *AfterChain*),按照各个操作变体在链中的具体位次依次逐个执行.值得注意的是,如果 *AroundChain* 为空,则执行 *AroundChain* 时,将会默认执行 *agt* 原先的操作 o;否则,将会执行 *AroundChain* 中第 1 个操作变体(可以理解为 *AroundChain* 中第 1 个操作变体重载(override)了原有的操作 o),然后转而执行 *AfterChain*,并最终将执行结果返回.

3 上下文感知增强过程以及运行支撑环境

上一节介绍了上下文感知 agent 模型,本节则进一步介绍利用该模型向既有 MAS 应用系统引入上下文感知能力的基本过程以及底层运行支撑环境的主要组成部分.

3.1 MAS系统上下文感知增强过程

目前,已有的 agent 系统绝大多数都是基于对象技术(OOP)的(例如,Aglet(http://aglets.sourceforge.net/), JADE(http://jade.tilab.com/)都是基于 Java 搭建的).一个 MAS 应用程序往往是由多个 agent 类所构成的;在运行时刻,众多 agent 实例相互通信、共同协作完成既定的计算任务;而 agent 的操作与接口也往往是依赖于对象方法(method)实现的.为此,下文中都是在面向对象技术(OOP)的基础上来讨论 agent 的,并将 agent 的操作称为操作方法(operation method).

如图 3 所示,对既有 MAS 应用系统进行上下文感知增强的过程可以大致划分为以下 4 个阶段:

- 既有 MAS 应用的自适应转换:针对一个既有的 MAS 应用系统,指定需要增添上下文感知能力的 agent 类的列表 L_{cls};针对每一个 agent 类 $A\in L_{cls}$,利用代码植入技术自动生成与之对应的具有上下文感知能力

的 agent 类.

- 操作变体实现:针对每一个 agent 类 $A \in L_{cls}$ 的某一个操作方法 $m()$,开发一系列的操作变体.
- 相关情境定义:针对转换后的应用系统,开发人员定义一系列该系统所涉及到的情境.
- 上下文感知策略(context-aware policy)规约描述:在上述操作变体以及情境定义的基础上,进一步定义一系列的上下文感知策略.

值得注意的是,在这 4 个阶段中,第 1 个阶段是后续 3 个阶段的基础,需要在系统运行前完成;而后续 3 个阶段则可以在系统运行时刻进行,从而实现 agent 上下文感知行为的动态添加与删除.接下来,我们详细介绍这 4 个阶段.

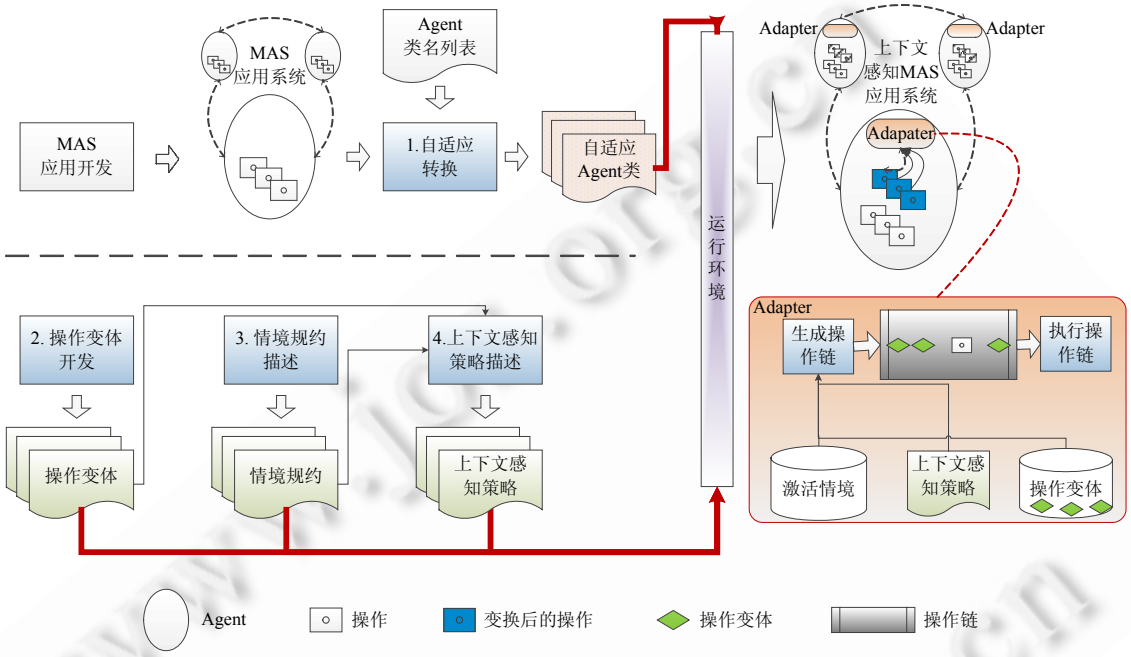

Fig.3 Process for enhancing a MAS with context-awareness

图 3 MAS 应用系统上下文感知增强过程

3.1.1 既有 MAS 应用的自适应转换

针对一个 MAS 应用系统,本阶段的主要任务是对开发人员所指定的一系列 agent 类进行自动扩展和转换,获得能够根据情境调节自身行为的 agent 类.

为了实现上文所述的上下文感知 agent 模型中的$\mathcal{S},\mathcal{V},\mathcal{P}$这 3 个部分,我们为 agent 引入一个 adapter(如图 3 所示),集中维护 agent 的上下文感知行为:adapter 维护 agent 所关注的情境$\mathcal{S}$及其运行时刻状态,统一管理 agent 所加载的上下文感知策略$\mathcal{P}$以及操作变体$\mathcal{V}$;adapter 拦截对原有的操作方法的调用,根据当前的情境信息以及所加载的上下文感知策略选择操作变体、创建并执行操作链.

具体而言,假定需要对一个名为 A 的 agent 类进行转换,整个转换过程包括两个主要部分:1) 对应 Adapter 类的生成;2) 对指定 agent 类的扩展.

- Adapter 类的生成

如图 4 所示,给定一个待转换的 agent 类 $A \in L_{cls}$(类名为 A),首先为其生成一个 *A_Adapter* 类.类似于 TRAP/J[27]在第 1 阶段所生成的元(meta-level)类,*A_Adapter* 负责在运行时刻动态决定 A 类某个操作的具体实现.*A_Adapter* 类是抽象类 *Adapter* 的子类;抽象类 *Adapter* 具有 3 个主要的域(fields):*agent*,*activeSits*,*ctxPolicies*.其中,*agent* 域指向一个转换后得到的上下文感知 agent 实例,*activeSits* 用于记录 *agent* 所关注的情境中处于激

活状态的那部分情境(即 $act(s,\mathcal{S})$),*ctxPolicies* 则记录 *agent* 所加载的上下文策略集合$\mathbb{P}$(一旦$\mathbb{P}$确定,同时也就决定了 *agent* 所关注的情境的集合$\mathcal{S}$).*Adapter* 实现 *I_SitListener* 接口,可以向一个 *CtxManager* 实例注册所关注的情境信息;*invokeMethod*()方法则是具体实现并执行上下文感知操作的方法,它根据当前激活的情境上下文信息 *activeSits* 以及被调用的方法创建一个操作链(*OperationChain*)对象 *chain*,进一步调用 *chain* 的 *execute*()方法,并将运行结果作为最终结果返回.

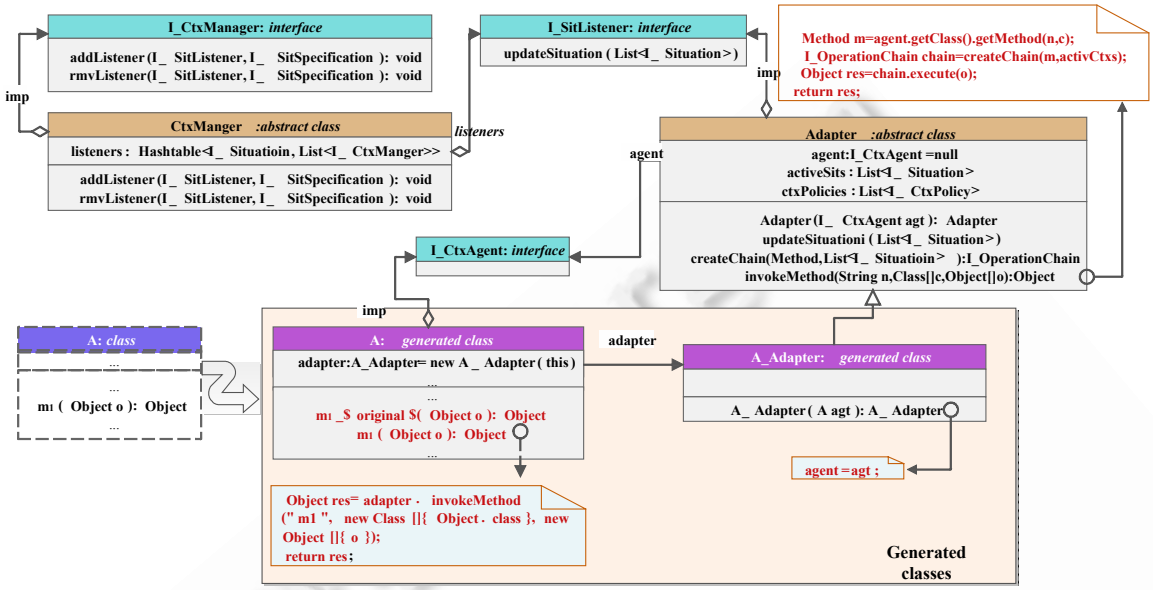

Fig.4 Transform a non-context-aware agent to be a context-aware one

图 4 将一个 agent 转换为具有上下文感知能力的 Agent

- 对指定 agent 类的转换与扩展

给定一个带转换的agent类$A\in L_{cls}$,利用代码植入技术对其进行转换和扩展.如图4所示,具体包括以下步骤:添加成员变量 *adapter*(*adapter* 的类型为 *A_Adapter*);我们用$\mathcal{O}(A)$表示 agent 类 *A* 的所有操作的集合(体现为 *A* 类的具体操作方法的集合).类似于 TRAP/J 的自动生成的 wrapper-level 类,针对 *A* 类所定义 agent 的每一个操作方法(即 $m\in\mathcal{O}(A)$),创建并植入一个新的方法 *m*′,该方法与 *m* 除了名字不相同之外,其他部分实现完全一致(如图4 所示,在实际实现中,我们规定 *m*′的方法名为 *m* 的方法名后添加"_$original$");之后,修改原有 *m* 的具体实现,使得所有对其调用都转发至其 *adapter*(调用 *adapter* 的 *invoke_method*()方法).

整个转换和扩展过程是通过上下文感知 agent 生成器(context-aware agent generator)自动完成的.至此,*A* 类被扩展和转换成为一个具有上下文感知能力的新版本**,具备了根据情境调节自身行为的能力,但其具体的上下文感知行为需要进一步通过定义操作变体、情境以及上下文感知策略来实现.

3.1.2 操作变体定义

针对列表中所列的某个 agent 类 $A\in L_{cls}$ 的某一个操作(基方法)$m\in\mathcal{O}(A)$,开发人员开发一系列的操作变体(operation variant).此处的操作变体与 COP[26]中的 partial method 类似:针对 *A* 类的某一个操作 $m\in\mathcal{O}(A)$的操作

** 为了加以区别,我们用 $\hat{A}$ 表示 *A* 转换后的得到的上下文感知新版本($\hat{A}$ 和 *A* 具有相同的类名,并且从功能上看,我们认为两者具有相同的操作(方法)集合,即 $\mathcal{O}(A)=\mathcal{O}(\hat{A})$).

变体 *ov* 是与 *m* 具有相同签名(signature)的方法(变体 *ov* 的返回类型可以为空,或者与 *m* 具有相同的返回类型[***]),我们称 *ov* 与 *m* 是兼容的,或称 *ov* 是 *m* 的一个操作变体.当在运行时刻 *m* 被调用时,系统将根据当前的情境以及上下文感知策略来动态组合这些变体,最终完成此次调用的具体执行.如图 5 所示,在我们的原型系统中,每一个操作变体是通过创建一个全新的类(作为抽象类 *OperationVariant* 的子类)而实现的.该子类仅实现 1 个与基方法具有相同签名的方法.沿用前面的表示方式,我们用 $\mathbb{V}(m)$表示所有与操作方法 *m* 兼容的操作变体的集合.

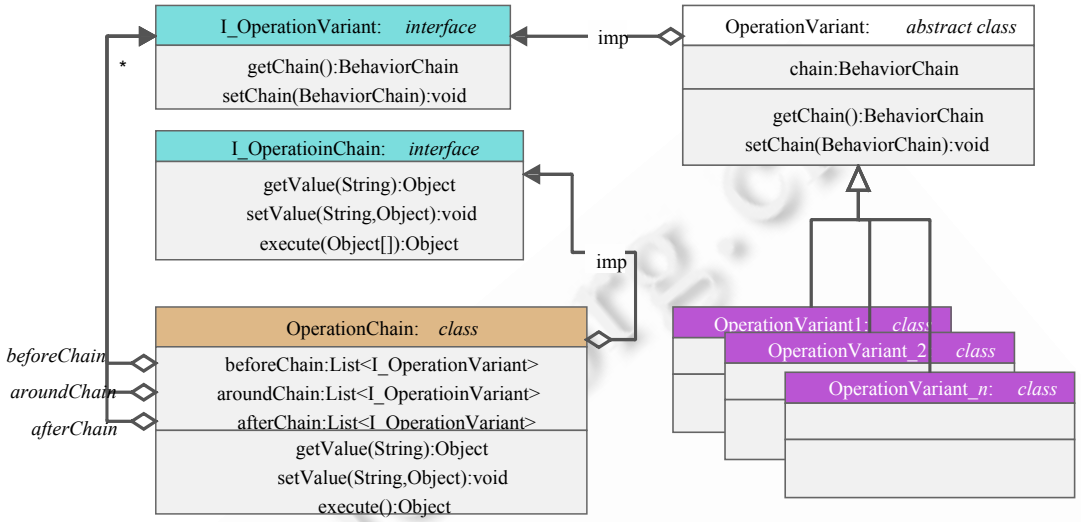

Fig.5 Operation variant and operation chain

图 5 操作变体以及操作链

3.1.3 情境规约定义

针对上阶段转换后的具有上下文感知能力的 MAS 系统,开发人员需要进一步定义一系列该系统所涉及到的情境(即 $\mathbb{S}(\mathcal{A})$).需要注意的是,针对指定的情境,有时候需要进一步开发、部署相应的软硬件传感器,以获取相关的原始感知信息.

3.1.4 上下文感知策略规约描述

在上下文感知策略规约描述阶段,开发人员需要定义一系列的上下文感知策略.这些策略将情境、agent 的操作以及操作变体联系起来,为运行时刻操作的动态组装和执行提供指导.具体而言,要定义一个上下文感知策略,系统开发人员或管理人员除了给定 $cp\doteq\langle o,s,ov,t,p,l_b\rangle$之外,还应进一步限定该策略所约束的 agent 对象.因此,此处一个具体的策略定义形如 $\langle\hat{C},set,\langle o,s,ov,t,p,l_b\rangle\rangle$.其中,$o,s,ov,t,p,l_b$ 的含义与第 2 节所介绍的一样;而 $\hat{C}$ 表示一个具有自适应能力的 agent 类,$set\subseteq instance(\hat{C})$ 表示一个由所有 $\hat{C}$ 类实例所构成集合(即 $instance(\hat{C})$)的一个子集.$\hat{C}$ 同 *set* 共同给定了策略 $cp\doteq\langle o,s,ov,t,p,l_b\rangle$所约束的 agent 范围.

3.2 运行支撑环境

为了支持转换后得到的上下文感知 MAS 系统的运行,我们提供了一个运行支撑环境(如图 6 所示).运行支撑环境主要包括以下几个主要部分:情境管理器(situation manager,简称 SM)管理系统中所有情境定义.管理人员通过 SM 动态地加载、卸载应用系统所关注的情境定义.上下文管理器(context manager,简称 CM)维护系统所涉及的各种底层上下文信息(即获取运行时刻的快照 s)并进一步完成情境评估的功能.CM 可以(直接)通过

*** 特别地,为了保证数据的一致性,我们规定:如果 *cp.t=before*,则 *cp.ov* 的返回类型为 *void*;如果 *cp.t=around*,则 *cp.ov* 的返回类型必须与 *cp.o* 一致;如果 *cp.t=after*,则 *cp.ov* 的返回类型为 *void* 或者与 *cp.o* 一致.

传感器从环境中获取上下文信息,或者从其他上下文管理平台(间接)获取上下文信息.CM 统一了系统中上下文表示,为上下文的处理、分发和应用提供统一的基础.CM 根据 SM 中所加载的情境规约,提供情境评估功能并进一步为上下文感知 agent 推送其所关注的情境状态.操作变体管理器(operation variant manager,简称 OVM)负责管理所有操作变体的类文件,系统中的 agent 通过 OVM 来获取具体的操作变体类文件从而进一步创建操作变体实例的.上下文感知策略管理器(context-aware policy manger,简称 CPM)则为系统中所有的上下文感知策略提供统一的管理支持.管理员通过 CPM 动态地添加、删除系统中某个(或某组)agent 所加载的上下文感知策略;Agent 管理器(agent manager,简称 AM)则建立在前面 4 个构件上,提供了对系统中运行的所有上下文感知 agent 的管理功能.通过 AM(以及上述 4 个构件),能够在运行时刻动态地添加、删除、替换操作变体、情境规约以及上下文感知策略来修改某个(或某一组)agent 的上下文感知行为.

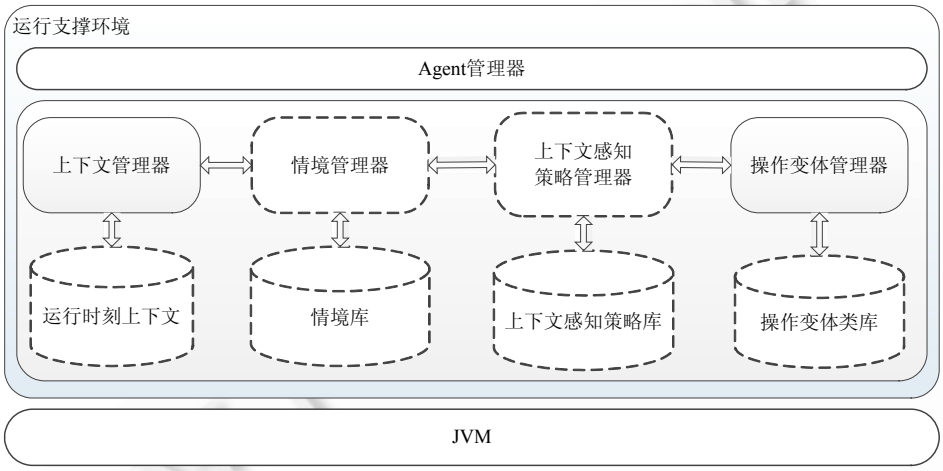

Fig.6 Runtime environment

图 6 运行支撑环境

4 原型系统与示例应用

在前面几节内容的基础上,为了展示本文所提出的方案的可行性,本节我们介绍一个原型系统的关键部分实现,并基于该系统实现对第 1 节所介绍的简单电子商务应用的上下文感知增强.

4.1 原型系统实现

在原型系统中,我们利用 JavAssist[28]所提供的字节码植入技术实现了上下文感知 agent 生成器(context-aware agent generator),从而在既有应用源码不可得的情况下,自动实现第 3.1.1 节所述的对指定 agent 类的转换、扩展操作.此外,我们利用 Jena 作为底层运行环境中上下文管理器(CM)的推理器实现,统一完成系统中上下文的推理,即情境评估操作.进一步地,情境规约也是基于 Jena 的正向推理规则(forward rule)进行定义的.具体而言,一个情境规约形如

$$Situation := sitation\_name : term,...term \tag{6}$$

此处,*sitation_name* 表示该情境的名,*term* 则和 Jena 规则中定义中的 *term* 一样(http://jena.apache.org/documentation/inference/index.html),表示一个具体的 RDF 三元组或者是 Jena 内置的一个函数.底层运行环境在加载情境规约时,自动将其转换为一条 Jena 正向推理规则(参见公式(7)),并将其加载至 CM 的 Jena 推理器.

$$term,...term \rightarrow sitation\_name \tag{7}$$

我们通过定义感知策略定义文件(context-aware policy definition file)来描述具体的上下文感知策略.具体而言,一个策略文件是一个 XML 文档,其具体结构可由图 7 所示的 Schema 文件来刻画.一个感知策略文件包括一个 *Ctx_Policies* 类型的根元素 *policies*,该元素由一系列的策略(*Ctx_Policy*)类 *policy* 元素所构成.一个 *policy*

定义一条具体的上下文感知策略.与第 3.1.4 节所介绍的上下文感知策略逻辑结构对应,定义一个具体的 *Ctx_Policy* 实例,我们需要给出该策略所约束的 agent 类型名(*className*)、所约束的 agent 实例集合(用 *AgentList* 类元素 *agentList* 刻画)、基方法的方法签名(利用 *OperationMethodType* 类型元素刻画)、卫士情境的名称(*sitationName*)、策略的优先级(*priority*)、编织类型(*weavingType*)、操作变体类(*OVClass*)以及屏蔽列表(*blockList*).

```
<?xml version="1.0" encoding="UTF-8"?>
- <schema targetNamespace="http://moon.nju.edu.cn/Jena_Ctx"
elementFormDefault="qualified" xmlns:tns="http://moon.nju.edu.cn/Jena_Ctx"
xmlns="http://www.w3.org/2001/XMLSchema">
    <element type="tns:Ctx_Policies" name="policies"/>
  - <complexType name="Ctx_Policies">
      - <sequence>
          <element type="tns:Ctx_Policy" name="policy" maxOccurs="unbounded"
            minOccurs="0"/>
        </sequence>
    </complexType>
  - <complexType name="Ctx_Policy">
      - <sequence>
          <element type="string" name="className"/>
          <element type="tns:AgentList" name="agentList"/>
          <element type="tns:OperationMethodType" name="operationMethod"
            maxOccurs="1" minOccurs="1"/>
          <element type="string" name="situationName" maxOccurs="1" minOccurs="1"/>
        - <element name="weavingType" maxOccurs="1" minOccurs="1">
            - <simpleType>
                - <restriction base="string">
                    <enumeration value="before"/>
                    <enumeration value="around"/>
                    <enumeration value="after"/>
                  </restriction>
              </simpleType>
          </element>
          <element type="int" name="priority" maxOccurs="1" minOccurs="1"/>
        - <element name="OVClass" maxOccurs="1" minOccurs="1">
            - <complexType>
                - <sequence>
                    - <element name="initialization" maxOccurs="unbounded"
                    minOccurs="0">
                      - <complexType>
                          - <sequence>
                              + <element name="arg" maxOccurs="unbounded"
                                minOccurs="0">
                            </sequence>
                        </complexType>
                    </element>
                  </sequence>
                  <attribute type="string" name="className"/>
              </complexType>
          </element>
          <element type="tns:BlockingList" name="blockingList" maxOccurs="1"
            minOccurs="0"> </element>
      </sequence>
    </complexType>
  + <complexType name="OperationMethodType">
  + <simpleType name="ParameterList">
  + <complexType name="AgentList">
  + <simpleType name="BlockingList">
</schema>
```

Fig.7 XML schema for a context policy file

图 7 上下文感知策略文件的 XML schema

值得一提的是,*AgentList* 具有一个属性 *type*∈{"*all*","*some*"}和一个元素 *agents*.当 *type*="*all*"时,表示当前策略约束所有 *className* 所指定 agent 类的实例,不必关心其元素 *agents* 是否存在或取值如何;当 *type*="*some*"时,表示当前策略仅约束 *agents* 元素所指定的一组(*className* 所指定类型的)agent 实例(如果 *agents* 此时为空列表,则该策略不起任何作用).

此外,操作链 *OperationChain* 的 *getValue*()及 *setValue*()方法(通过这两个方法,操作变体可以访问 agent 的各个域(field))、操作链的执行(即对各个操作变体的调用)等操作(如图 5 所示)都是依赖 Java 反射机制而实现的.

4.2 示例应用

本节展示如何使用本文提出的技术手段来向第 1 节所介绍的电子商务 MAS 系统引入上下文感知能力的具体过程.我们事先基于 Aglet 系统(本文提出的技术框架具有通用性,能够适用于基于 java 的其他 MAS 平台)完成对 *SellerAgent*,*BuyerAgent* 以及 *MarketAgent* 的初始开发(即完成系统应用逻辑的开发),得到编译后的字节

码.下面我们展示如何在此字节码基础上引入上下文感知能力,以满足需求 *R*-1(独立地引入上下文感知逻辑).

首先,明确所需引入上下文感知能力的 Agent 类列表 L_{cls}.此处,我们仅需对 *SellerAgent* 类进行上下文感知增强转换,所以 L_{cls}={*SellerAgent*}(其实,此处可以对 *SellerAgent*,*BuyerAgent* 以及 *MarketAgent* 同时进行转换,不会影响系统的运行);利用上下文感知 agent 生成器对 *SellerAgent* 类进行上下文感知增强转换,得到全新的具有上下文感知能力的 *SellerAgent* 类以及对应的 *SellerAgent_Adapter* 类.然后,按照上节所约定的情境规约格式定义相关的情境规约"Holiday":

```
Holiday:now(?x),ge(?x,'2012-05-1Z'^^http://www.w3.org/2001/XMLSchema#date),
        ge('2012-05-4Z'^^http://www.w3.org/2001/XMLSchema#date,?x),
Holiday:now(?x),ge(?x,'2012-10-1Z'^^http://www.w3.org/2001/XMLSchema#date),
        ge('2012-10-8Z'^^http://www.w3.org/2001/XMLSchema#date,?x).
```

随后,我们进一步为 *SellerAgent* 的 *getPrice*()方法定义在 Holiday 情境下的操作变体以实现具体的上下文感知行为(即"打九折").为此,我们定义一个名为 *DiscountOV* 的操作变体(如图 8 所示).*DiscountOV* 继承 *Operation_Variant* 抽象类,具有一个属性 *dc_rate* 以及唯一的一个方法 *getPrice*():*dc_rate* 用于记录具体的打折力度,而 *getPrice*()则是操作变体的具体实现部分.运行时刻,该方法利用 *getOwner*()获取该变体所属的操作链,并进一步利用该操作链提供的 *getInter_res*()获取操作链执行过程中的中间结果(即操作链中位于当前变体之前的其他变体执行完毕后的待返回结果).最终对该中间结果进行处理并返回处理结果.

```
1. public class DiscountOV extends Operation_Variant {
2.     double dc_rate=0.9;
3.     public double getPrice(int itemNO) {
4.         double res=(double)this.getOwner().getInter_res();
5.         res=res*this.dc_rate;
6.         return res;
7.     }
8. }
```

Fig.8 Operation variant *DiscountOV*

图 8 操作变体 *DiscountOV*

最后,我们定义一个上下文感知策略 cp_1(如图 9 所示),将 Holiday 情境与 *DiscountOV* 操作变体结合起来.在该策略定义中,*className=emarket.SellerAgent*,*AgentList.type=all*,表明本策略约束所有的 *SellerAgent* 类实例;*weavingType=after*,表明 *DiscountOV* 操作变体应该在 *SellerAgent* 的原有操作 *getPrice*()执行之后运行;此外,策略的优先级设定为 5.

```
<?xml version="1.0" encoding="UTF-8"?>
- <tns:policies
xsi:schemaLocation="http://moon.nju.edu.cn/Jena_Ctx ../../../
" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:tns="http://moon.nju.edu.cn/Jena_Ctx">
  - <tns:policy>
      <tns:className>emarket.SellerAglet</tns:className>
      <tns:agentList type="all"/>
    - <tns:OperationMethod>
        <tns:methodName>getPrice</tns:methodName>
        <tns:parameterList>int</tns:parameterList>
      </tns:OperationMethod>
      <tns:situationName>Holiday</tns:situationName>
      <tns:weavingType>after</tns:weavingType>
      <tns:priority>5</tns:priority>
    - <tns:OVClass className="emarket.ov.DiscountOV">
      - <tns:initialization>
          <tns:arg type="double">0.9</tns:arg>
        </tns:initialization>
      </tns:OVClass>
      <tns:blockingList/>
    </tns:policy>
</tns:policies>
```

Fig.9 A context-aware policy definition file

图 9 一个的感知策略定义文件

至此,我们完成了原有应用的上下文感知增强转换和指定上下文感知行为的开发,接下来则将上述文件部署到底层运行平台来运行(如图 10(a)所示).图 10(b)展示了未添加情境定义以及感知策略时的运行情况:由于没有加载任何策略,当 *SellerAgent* 的 *getPrice*()操作方法执行时,系统为其创建一个空的操作链;当执行该操作链时则直接执行 *SellerAgent* 原有的 *getPrice*()方法(此处已经改名为 *getPrice_$Orig$*());与之相对应,图 10(c)展示了添加"Holiday"情境定义以及感知策略(如图 9 所示)之后的运行情况:底层运行平台首先利用 Jena 推理器得到当前的情境信息"Holiday"并通知 *SellerAgent*;当执行 *SellerAgent* 的 *getPrice*()方法时,系统根据当前的情境以及加载的策略为其创建一个操作链,该操作链仅包含 1 个编织类型为 *after* 的操作变体 *DiscountOV*.当执行该操作链时则首先执行 *SellerAgent* 原有的 *getPrice*()方法(即 *getPrice_$Orig$*()),然后进一步执行 *DiscountOV* 的 *getPrice*()方法,最终得到打折后的商品价格(如图 10(d)所示).

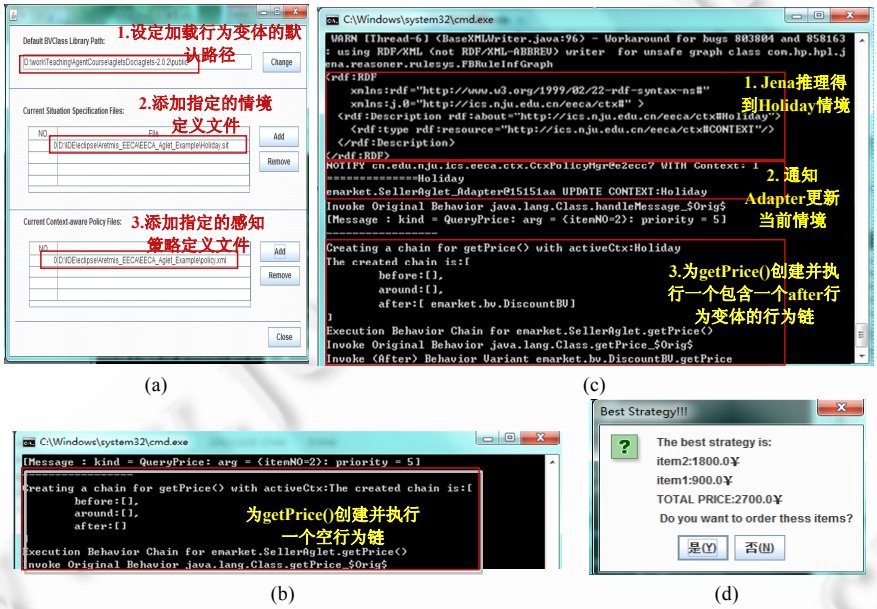

Fig.10 A demo application enhanced with context-awareness

图 10 上下文感知增强的示例应用

需求 *R*-1 的实现展示了如何对一个既有 MAS 系统进行上下文增强的过程(自适应转换,情境、操作变体以及上下文感知策略的开发与部署).为了展示如何对转换后的系统动态地调整上下文感知行为,接下来我们实现需求 *R*-2.

由于前面我们已经完成对 *SellerAgent* 的上下文感知转换,此处我们仅需要针对 *R*-2 定义相应的情境"Weekend"以及策略 cp_2(操作变体仍为 *DiscountOV*,不过将 *dc_rate* 设为 0.95),并利用运行支撑环境动态加载新的情境和策略即可在线更新 *SellerAgent* 上下文感知行为.不过,此时两个策略之间没有屏蔽关系,两者是共存的(即存在既是周末又是假期的情况,此时商品折扣为两者的乘积 0.9×0.95=0.855).为了避免这一情况,我们可以将 cp_2 的优先级设定为 4(或更低),并设定 cp_1 的 l_b={*Weekend*}.这样,当既是周末又是假期的情况发生时,仅有 cp_1 被激活,而 cp_2 则被屏蔽.

4.3 性能评估

通过上下文感知增强的 agent 在执行操作方法时需要根据当前的情境动态地组装和执行操作链;而操作链的组装和执行必然带来额外的开销.本节我们通过一个实验来展示创建、执行操作链带来的开销.

本实验仅涉及 1 个 *HelloAgent* 类,该类仅有 1 个 *printHello*()方法(直接在屏幕打印"Hello World!").我们对

HelloAgent 类进行上下文感知增强转换,并针对 *printHello*()方法定义一个操作变体 *EmptyOV*.该变体的 *printHello*()方法不执行任何代码,为一个空方法.进一步地,每次实验我们定义 *n* 个策略(*n*=0,1,…,10).每个策略的配置如下:

- *className=HelloAgent*;
- *AgentList.type=all*;
- *OperationMethod=printHello*();
- *weavingType=before*;
- *situationName=true*(表示一个永远评估为真的情境);
- *OVClass.className=EmptyOV*;
- *priority=i*(*i*=1,…,*n*).

按照上述配置,针对每一个 *n*,在执行 *printHello*()方法时,会生成并执行一个包含 *n* 个操作变体的操作链.

我们执行 *printHello*()方法 10 000 次并记录下累计的执行时间(运行的环境配置如下:内存(4G),CPU(Intel Core i5 2.53GHz),OS(WIN7)),并进一步获取每次执行的平均时间(图 11 展示了实验的结果).为了比较,我们还记录了执行原有(未转换前的)*HelloAgent* 的 *printHello*()的时间(如图 11 中虚线所示,大约 11.6μs).

由图 11 不难看出,操作链的创建和执行的确引入了额外的时间开销,且此开销随着操作链长度的增加而线性增长:每增加一个操作变体引入大约 1.8μs 额外的执行时间.

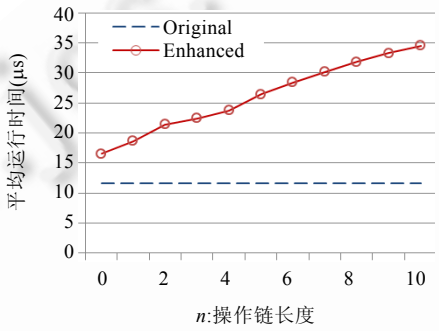

Fig.11 Execution time of operation chains with different lengths

图 11 不同长度操作链执行时间

尽管会带来一定的开销,但本文所提出的上下文感知增强框架为既有 MAS 系统(特别是在系统源码不可获取的情况下)的上下文增强以及全新上下文感知 MAS 系统的开发提供了一套通用的、行之有效的途径.只要目标 MAS 系统对执行效率没有苛刻的要求,本文的技术框架就能为系统的上下文感知增强提供切实可行的支持.

5 相关工作

由于 agent 技术与生俱来的自主性、移动性等特征,使其尤为适应面向开放、动态、难控的计算环境的应用系统的开发.基于 agent 的上下文感知计算研究工作大致可以简单划分为以下两类:基于 agent 的上下文感知应用系统与基于 agent 的上下文感知框架.基于 agent 的上下文感知应用系统研究主要是利用上下文信息以及自主、移动、协同的 agent 来实现一个个具体的上下文感知应用系统(例如,文献[5]中利用 agent 实现了具有一个上下文感知能力的教育游戏系统,NAMA[6]则是一个基于 MAS 的个性化提醒系统).

而基于 agent 的上下文感知框架研究工作则关注于以 agent 作为底层支撑技术,为上层上下文感知应用系统提供一个统一的、通用的支撑平台或编程框架[9,17−25].它们针对上下文感知的某一个(或多个)环节给出了一个统一的框架或平台,为上下文感知自适应 MAS 系统的设计和开发提供了不同程度、不同方面的支持.例如,CAMPS[23]为智能环境(smart space)中应用使用上下文信息提供了一套基于 agent 的中间件平台.CoBrA[17],

ACAI[24]]以及 Artemis-FollowMe[20]均将普适计算环境视为由一系列的 domain 所构成,并基于 agent 技术提供了一个层次式的上下文管理框架,统一负责系统中上下文信息的获取、处理以及分发.Gaia[18]针对普适计算环境提供了一个中间件平台,该平台一方面使得环境中的各个 agent 能够方便地获取环境中的各种上下文信息,另一方面为 agent 交互提供统一的支持.AmbieAgents[19],AmIciTy[25]基于 agent 提供了一套基础支撑框架以支持基于上下文的信息共享.EgoSpace[21]为 ad hoc 环境下的上下文感知应用开发和运行提供了一个面向 agent 的基于元组空间(tuple space)协同中间件平台:通过定义一系列的视图(view),一个 agent 可以从其周围的其他 agent 处获取所关注的上下文信息,从而进一步调节自身行为.环境驱动软件模型[9]提出了一套情境驱动的基于软件 agent 开放协同的网构软件(internetware)模型,为面向开放、动态、难控的计算环境的软件系统的设计和开发提供了有效的支持.MDAgent[22]则利用软件 agent 来封装应用系统的各个组成构件,从而实现了应用系统跟随移动的用户在普适环境中的无缝迁移.

与上述研究工作不同,本文从软件工程角度出发,依照关注分离原则,将系统的应用逻辑与上下文感知逻辑解耦合,提出一套开发上下文感知 MAS 应用系统的技术框架和方法体系.特别地,本文提出的方法利用 JavAssist[28]所提供的代码植入技术,对既有非上下文感知的 MAS 应用系统进行自动转换和扩展,即便在既有系统源码不可获取的情况下,仍可方便地引入上下文感知能力;而且针对转换后的系统,可以在运行时刻在线调整其上下文感知行为,实现上下文感知行为独立的、动态的、增量式的开发.

此外,为了进一步支持上下文感知自适应 MAS 应用系统的开发,还有一部分研究工作尝试通过 Agent 在运行期间转换角色或类型来实现系统的自适应[29−32].特别地,文献[29,30]借鉴组织学和社会学的概念和思想,通过让 Agent 动态绑定不同的角色类来达到适应环境变化的目标.类似本文的上下文感知策略,文献[29,30]提出了一个基于动态绑定机制的自适应策略描述语言 SADL,为刻画 Agent 的自适应行为提供了一个结构良好的、简单的工具.两者的目的都是根据关注分离原则,将系统的业务逻辑与上下文感知(或自适应)逻辑分离,实现上下文感知(或自适应)逻辑的独立开发和动态加载.其不同之处在于,本文的上下文感知策略刻画的是操作变体如何被编织到最终操作链并执行,而 SADL 描述的自适应策略刻画的是 Agent 如何根据当前状况调节自身的角色.最重要的是,二者的侧重点不同,文献[29,30]更加关注如何独立而方便地刻画 Agent 的自适应行为,从而简化复杂自适应系统的开发和维护;而本文的出发点则是如何有效地对既有非上下文感知的 MAS 系统进行上下文感知增强操作.

面向上下文程序设计(context-oriented programming,简称 COP)[26]是对面向对象程序设计(object-oriented programming,简称 OOP)的扩展,它为上下文感知应用的系统设计提供了一套全新的编程范式.目前,已经有不少工作尝试对各种主流编程语言进行扩展以提供 COP 的支持[26,33].layer 是 COP 中极为重要的一个基本概念:一方面,layer 作为系统中上下文信息的一个表示;另一方面,一个 layer 同时又是封装系统上下文感知行为的一个结构.当利用上述这些 COP 语言[33]进行编程时,程序员可以声明不同的 layer,以表示目标系统可能会使用到的上下文信息,每一个定义好的 layer 可以在系统运行过程中动态地激活(activate)或去活(deactivate);在此基础之上,针对某一给定的类 class A 的某一种方法 m()(称为基方法(base method)),程序员(利用上述语言提供的设施)可以进一步定义该方法在某一 layer 被激活的情况下的偏方法(partial method).当某一 class A 对象 o 的 m()方法在系统运行过程中被调用时,系统会根据此时被激活的所有 layer 来查找相对应的各个偏方法,并将这些偏方法同原先的基方法 m()动态组合以完成此次方法调用的最终行为(详情请参见文献[26]).COP 将上下文作为程序设计中的一个显式表示和操作的成分,为上下文感知应用系统的设计和开发提供了直观、简洁的支持;不过,由于现有提供 COP 支持的语言[33]都引入了全新的语言成分,这些语言比较适合全新的上下文感知应用的开发.

我们借鉴了 COP[26]中的偏方法(partial method)的概念与动态组合调用行为的思想:文中的操作变体(operation variant)类似于 COP[26]中的偏方法,当 agent 的某个操作方法被调用执行时,将会根据运行时刻的情境信息动态组合各个操作变体,从而实现 agent 的上下文感知行为.另外,与上述 COP 语言[33]扩展全新语言成分不同,本文遵循关注分离原则,提供了一套独立的申明机制,应用开发人员以及管理员可以独立地定义系统所需使用的情境以及上下文感知策略,上下文感知逻辑与系统业务逻辑进一步解耦合,不仅适用于对既有 MAS 系统进

行上下文感知增强,而且也为开发全新的上下文感知 MAS 系统提供了有效的途径.

如何向既有遗产软件引入自适应能力,是自适应软件的一个重要的研究领域,这方面如今已有不少的工作[27,34,35].TRAP/J[27]利用反射(reflection)以及面向方面程序设计(aspect-oriented programming,简称 AOP)[36]技术提供了一个通用的框架和软件工具,使得开发人员能够在不显式修改应用源码或修改底层 JVM 的前提下向既有 JAVA 应用系统添加自适应行为.简单来说,TRAP/J[27]包括两个阶段:第 1 阶段,TRAP/J 在编译时刻将一个既有应用系统转换成一个 adapt-ready 应用:开发人员选择一组需要添加自适应功能的类,针对每一个类,TRAP/J[27]自动为其生成一个 aspect 文件、一个包装(wrapper-level)类和一个元(meta-level)类,并利用 AspectJ 对应用的源代码以及上述生成的文件进行编织(weaving),最终得到 adapt-ready 的应用系统;第 2 阶段,在运行时刻利用先前生成的包装(wrapper-level)类和元(meta-level)类,动态向应用系统引入新的行为.然而,由于 AspectJ 需要对应用的源码进行编织,TRAP/J[27]只能适用于应用源码可获取的情况.此外,TRAP/J 针对一个用户所指定的类所生成的 wrapper-level 类实际上是由用户所指定类的一个子类来实现的,而不会影响该类的其他子类,因此,当要求一个类的所有实例(包括其子类的实例)都具有自适应能力时,必须对该类的所有子类都生成对应的 aspect、包装类和元类.与 TRAP/J 利用 AOP 不同,JavAdaptor[35]则利用 JavAssist[28]提供的(类加载时刻)字节码植入的能力、类重命名等技术实现对 java 语言所编写的应用程序的动态更新.

通过本文提供的上下文增强机制转换得到的 agent-adapter 结构与 TRAP/J[27]中的包装(wrapper-level)类和元(meta-level)类相似,都是将方法的调用拦截并转交给 adapter(或 meta-level 的对象)进行动态处理.其不同在于,TRAP/J 中 meta-level 类对拦截的方法调用是直接转发给一个具体 delegate 对象来实现的,而本文的 adapter 则是根据调用时刻的情境信息动态的从一组操作变体中选取、组装得到一个操作链,并进一步运行该操作链.此外,TRAP/J 利用 AspectJ 生成包装(wrapper-level)类和元(meta-level)类,需要获得原有应用系统的源码;而本文则和 JavAdaptor[35]一样,利用 JavAssist[28]提供的字节码植入能力,可以在源码不可获取的情况下对既有应用系统进行转换,更加灵活、简便.agent-adapter 的结构使得对既有 MAS 系统进行一次离线的转换之后,就可以在运行时刻在线动态地调整指定 agent 的上下文感知行为.

6 总结与展望

开放、动态、难控的计算平台要求软件系统能够根据系统自身及其环境的状态信息(以及状态改变)动态地调节自身的行为,即具备一定的上下文感知能力.然而,现有的工作并未就如何向既有的 MAS 应用系统引入上下文感知能力提出有效的解决方案.本文结合面向上下文程序设计技术(COP)[26]、反射技术(reflection)以及代码植入技术,提出了一套向既有 MAS 应用系统引入上下文感知能力的途径和运行支撑框架.通过该框架,开发人员可以在既有应用源码不可得的情况下,自动将既有非上下文感知的 MAS 系统转换为(扩展为)具有上下文感知能力的 MAS 应用.此外,利用底层运行支撑环境提供的支持,系统管理员可以在系统运行时刻动态地添加、删除系统中指定 agent 的上下文感知行为.最后,本文所提出的方法充分体现了关注分离设计原则,将系统的关键应用逻辑与上下文感知逻辑的设计和开发解耦合,不仅适用于向既有 MAS 系统引入上下文感知能力,而且支持开发全新的上下文感知 MAS 系统.

在现阶段的设计中,我们采用了类似于 COP[26]的设计,要求操作变体应与基方法具有相同的方法签名.在今后的工作中,我们将尝试在上下文感知策略中添加操作变体方法规约,从而进一步放松对操作变体的这些限制.目前原型系统中操作变体通过操作链提供的 *getValue*()以及 *setValue*()个方法来访问 agent 的各个域(field),不是十分方便和简洁.我们当前正在尝试结合 AOP 和 Java 反射机制实现方法变体能够像访问其自身的域(field)一样更加简洁地访问 agent 的各个域(field),从而统一编程的风格.目前,我们的框架中的 agent 能够根据情境主动(proactive)调节自身操作的能力(根据情境组装操作链),但外界情境的变化并不能直接触发 agent 执行特定的操作,即 agent 不具备对环境的反应能力(reaction).今后,我们将结合事件驱动模型以及环境驱动模型[8]进一步扩展上下文感知 agent 模型.今后我们将尝试进一步结合现有的软件动态更新技术(例如文献[34,35]),以期实现对处于运行状态下的 MAS 系统植入上下文感知能力.

References:

[1] Aylett R, Brazier F, Jennings N, Luck M, Nwana H, Preist C. Agent systems and applications. The Knowledge Engineering Review, 1998,13(3):1−7.

[2] Outtagarts A. Mobile agent-based applications: A survey. Journal of Computer Science, 2009,9(11):331−339.

[3] Sun R. Cognitive architectures and multi-agent social simulation. In: Lukose D, Shi Z, eds. Proc. of the Multi-Agent Systems for Society. LNAI 4078, Berlin, Heidelberg: Springer-Verlag, 2009. 7−21. [doi: http://dx.doi.org/10.1007/978-3-642-03339-1_2]

[4] Rogers A, David E, Jennings NR, Schiff J. The effects of proxy bidding and minimum bid increments within eBay auctions. ACM Trans. on the Web, 2007,1(2):572−581.

[5] Lu C, Chang M, Kinshuk, Huang E. Usability of context-aware mobile educational game. Knowledge Management & E-Learning, 2011,3(3):448−477.

[6] Kwon O, Choi S, Park G. NAMA: A context-aware multi-agent based web service approach to proactive need identification for personalized reminder systems. Expert Systems with Applications, 2005,29(1):17−32. [doi: http://dx.doi.org/10.1016/j.eswa.2005.01.001]

[7] Lu J, Tao XP, Ma XX, Hu H, Xu F, Cao C. Study on agent-based Internetware model. Science in China (Series E: Information Sciences), 2005,35(12):1233−1253 (in Chinese).

[8] Lu J, Ma XX, Tao XP, Cao C, Huang Y, Yu P. Study on environment-driven software model and technologies for Internetware. Science in China (Series F: Information Sciences), 2008,38(6):864−900 (in Chinese).

[9] Lu J, Ma XX, Tao XP, Cao C, Huang Y, Yu P. On environment-driven software model for Internetware. Science in China (Series F: Information Sciences), 2008,51(6):683−721. [doi: http://dx.doi.org/10.1007/s11432-008-0057-6]

[10] Weiser M. The computer for the 21st century. SIGMOBILE Mobile Computing and Communications Review, 1999,3(3):3−11. [doi: http://dx.doi.org/ 10.1145/329124.329126]

[11] Xu GY, Shi YC, Xie WK. Pervasive/Ubiquitous computing. Chinese Journal of Computers, 2003,26(9):1042−1050 (in Chinese with English abstract).

[12] Dey AK. Understanding and using context. Personal and Ubiquitous Computing, 2001,5(1):4−7. [doi: http://dx.doi.org/10.1007/s007790170019]

[13] Schilit BN, Adams N, Want R. Context-Aware computing applications. In: Proc. of the 1st Workshop on Mobile Computing Systems and Applications (WMCSA'94). Washington: IEEE Computer Society, 1994. 85−90. [doi: http://dx.doi.org/10.1109/MCSA.1994.512740]

[14] Salehie M, Tahvildari L. Self-Adaptive software. ACM Transa. on Autonomous and Adaptive Systems, 2009,4(2):1−42. [doi: http://dx.doi.org/10.1145/1516533.1516538]

[15] Soldatos JK. Software agents in ubiquitous computing: Benefits and the CHIL case study. In: Proc. of the Software Agents in Information. 2006.

[16] Cheverst K, Davies N, Mitchell K, Friday A. Experiences of developing and deploying a context-aware tourist guide: The GUIDE Project. In: Proc. of the 6th Annual Int'l Conf. on Mobile Computing and Networking (MobiCom 2000). Boston: ACM Press, 2000. 20−31. [doi: http://dx.doi.org/10.1145/345910.345916]

[17] Chen HL. An intelligent broker architecture for context-aware systems [Ph.D. Thesis]. Computer Science at the University of Maryland Baltimore County, 2003.

[18] Ranganathan A, Campbell RH. A middleware for context-aware agents in ubiquitous computing environments. In: Endler M, ed. Proc. of the ACM/IFIP/USENIX 2003 Int'l Conf. on Middleware (Middleware 2003). New York: Springer-Verlag, 2003. 143−161. [doi: 10.1007/3-540-44892-6_8]

[19] Lech TC, Wienbofen LWM. AmbieAgents: A scalable infrastructure for mobile and context-aware information services. In: Proc. of the 4th Int'l Joint Conf. on Autonomous Agents and Multi-Agent Systems (AAMAS 2005). New York: ACM Press, 2005. 625−631. [doi: http://dx.doi.org/10.1145/1082473.1082568]

[20] Ma J, Wang L, Xu JW, Tao XP, Lu J. Artemis-FollowMe: An agent-based middleware for mobile context-aware applications. In: Proc. of the 6th Int'l Conf. on Pervasive Computing and Applications. Port Elizabeth: IEEE, 2011. 139−145. [doi: http://dx.doi.org/10.1109/ICPCA.2011.6106493]

[21] Julien C, Roman G. Egocentric context-aware programming in ad hoc mobile environments. ACM SIGSOFT Software Engineering Notes, 2002, 27(6):21−30. [doi: http://dx.doi.org/10.1145/605466.605471]

[22] Zhou Y, Cao JN, Raychoudhury V, Siebert J, Lu J. A middleware support for agent-based application mobility in pervasive environments. In: Proc. of the 27th Int'l Conf. on Distributed Computing Systems Workshops (ICDCSW 2007). 2007. [doi: http://dx.doi.org/10.1109/ICDCSW.2007.12]

[23] Qin WJ, Suo Y, Shi YY. Camps: A middleware for providing context-aware services for smart space. In: Proc. of the 1st Int'l Conf. on Advances in Grid and Pervasive Computing. 2006. 644−653. [doi: 10.1007/11745693_63]

[24] Khedr M, Karmouch A. Acai: Agent-based context-aware infrastructure for spontaneous applications. Journal of Network and Computer Applications, 2005,28(1):19−44. [doi: http://dx.doi.org/10.1016/j.jnca.2004.04.002]

[25] Olaru A, Gratie C. Agent-Based, context-aware information sharing for ambient intelligence. Int'l Journal on Artificial Intelligence Tools, 2011,20(6):985−1000. [doi: http://dx.doi.org/10.1142/S0218213011000498]

[26] Hirschfeld R, Costanza P, Nierstrasz O. Context-Oriented programming. The Journal of Object Technology, 2008,7(3):125−151. [doi: http://dx.doi.org/10.5381/jot.2008.7.3.a4]

[27] Sadjadi SM, Mckinley PK, Cheng BHC, Stirewalt REK. TRAP/J: Transparent generation of adaptable Java programs. In: Meersman R, Tari Z, eds. Proc. of the CoopIS/DOA/ODBASE(2). Springer-Verlag, 2004. 1243−1261.

[28] Chiba S, Nishizawa M. An easy-to-use toolkit for efficient Java bytecode translators. In: Proc. of the 2nd Int'l Conf. on Generative Programming and Component Engineering (GPCE 2003). New York: Springer-Verlag, 2003. 364−376. [doi: 10.1007/978-3-540-39815-8_22]

[29] Hao XL, Dong MG, Mao XJ, Qi ZC. Design and implementation of a compiler for the self-adaptation strategy description language. Acta Electronica Sinica, 2009,37(B04):65−69 (in Chinese with English abstract).

[30] Dong MG, Mao XJ, Chang ZM, Wang J, Qi ZC. Running mechanism and strategy description language SADL for self-adaptive MAS. Journal of Software, 2011,22(4):609−624 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/3762.htm [doi: 10.3724/SP.J.1001.2011.03762]

[31] Wang J, Shen R, Zhu H. Towards an agent oriented programming language with caste and scenario mechanisms. In: Dignum F, Dignum V, Koenig S, Kraus S, Singh MP, Wooldridge M, eds. Proc. of the Int'l Joint Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2005). New York: ACM Press, 2005. 1297−1298. [doi: http://dx.doi.org/10.1145/1082473.1082741]

[32] Cabri G, Ferrari L, Leonardi L. Enabling mobile agents to dynamically assume roles. In: Proc. of the 2003 ACM Symp. on Applied Computing (SAC). Melbourne, 2003. 56−60. [doi: http://dx.doi.org/10.1145/952532.952546]

[33] Appeltauer M, Hirschfeld R, Haupt M, Lincke J, Perscheid M. A comparison of context-oriented programming languages. In: Proc. of the Int'l Workshop on Context-Oriented Programming. 2009. 1−6. [doi: http://dx.doi.org/10.1145/1562112.1562118]

[34] Bialek RP. Dynamic updates of existing Java applications [Ph.D. Thesis]. Department of Computer Science, Faculty of Science at University of Copenhagen Denmark, 2006.

[35] Pukall M, Kästner C, Cazzola W, Grebhahn A, Schröter R, Saake G. JAVADAPTOR–Flexible runtime updates of Java applications. Software: Practice and Experience, 2011,42(6):1−33.

[36] Kiczales G, Lamping J, Mehdhekar A, Maeda C, Lopes CV, Loingtier JM, Irwin J. Aspect-Oriented programming. In: Proc. of the European Conf. on Object-Oriented Programming (ECOOP). LNCS 1241, Springer-Verlag, 1997.

附中文参考文献:

[7] 吕建,陶先平,马晓星,胡昊,徐锋,曹春.基于 Agent 的网构软件模型研究.中国科学(E 辑:信息科学),2005,35(12):1233−1253.

[8] 吕建,马晓星,陶先平,曹春,黄宇,余萍.面向网构软件的环境驱动模型与支撑技术研究.中国科学(F 辑:信息科学),2008,38(6):864−900.

[11] 徐光祐,史元春,谢伟凯.普适计算.计算机学报,2010,26(9):1042−1050.

[29] 郝小雷,董孟高,毛新军,齐治昌.自适应 Agent 策略描述语言的设计及编译器的实现.电子学报,2009,37(B04):65−69.

[30] 董孟高,毛新军,常志明,王戟,齐治昌.自适应多 Agent 系统的运行机制和策略描述语言 SADL.软件学报,2011,22(4):609−624. http://www.jos.org.cn/1000-9825/3762.htm [doi: 10.3724/SP.J.1001.2011.03762]

马骏(1980－),男,贵州安顺人,博士生,讲师,CCF 会员,主要研究领域为上下文感知计算,软件 agent 技术,普适计算中间件.

陶先平(1970－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为对象技术,软件 agent 技术,中间件技术,普适计算技术.

朱怀宏(1959－),男,副教授,CCF 高级会员,主要研究领域为计算机软件.

吕建(1960－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为软件自动化,面向对象语言,环境和并行程序的形式化方法.