ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.18, No.3, March 2007, pp. 582−591
DOI: 10.1360/jos180582

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

软件 Agent 的一种面向对象设计模型*

黎建兴+, 毛新军, 束 尧

(国防科学技术大学 计算机学院,湖南 长沙 410073)

An Object-Oriented Design Model of Software Agent

LI Jian-Xing+, MAO Xin-Jun, SHU Yao

(School of Computer, National University of Defense Technology, Changsha 410073, China)

+ Corresponding author: Phn: +86-731-4573649, Fax: +86-731-4575802, E-mail: jxli@nudt.edu.cn

Li JX, Mao XJ, Shu Y. An object-oriented design model of software Agent. *Journal of Software*, 2007,18(3): 582−591. http://www.jos.org.cn/1000-9825/18/582.htm

Abstract: How to implement software Agent is a key problem to develop Agent-oriented programming languages and development tools. Aiming to solve this problem based on the mainstream OO (object oriented) technology, this paper first discusses the differences between Agent and object, and then presents an implemental architecture and behaviors decision algorithm of software Agent. The architecture and related algorithm are based on OO technology and the simplified improvement of BDI (belief desire intention) Agent model. An OO-based design framework of software Agent is also proposed by using the POAD (pattern-oriented analysis and design) method. The approach is helpful to clarify how to extend the OO method suitably for developing Agent-oriented programming languages and development tools.

Key words: software agent; implemental architecture; design framework; POAD (pattern-oriented analysis and design) method

摘 要: 怎样实现软件 Agent 是设计与开发面向 Agent 编程语言及工具的关键问题.为基于当前主流的面向对象技术来解决该问题,首先讨论了对象与 Agent 的主要区别,然后通过对 Agent 的 BDI 模型进行简化改进,提出了一种基于对象技术的软件 Agent 的实现体系结构及其内部行为自主决策算法.最后,基于该结构及算法并利用设计模式开发出了软件 Agent 的一种面向对象设计框架.该工作对于上述问题的解决,以及在现有成熟的面向对象技术基础上开发出软件 Agent 程序设计语言及其支撑环境具有基础指导意义.

关键词: 软件 Agent;实现体系结构;设计框架;POAD(pattern-oriented analysis and design)方法
中图法分类号: TP18 文献标识码: A

随着计算机网络技术的迅猛发展以及 Internet 的普及应用,对运行于网络分布环境中的各种应用软件系统的需求不断涌现,而这些需求中有很多都呈现出自主、主动适应、动态可伸缩、开放可成长等复杂的特征.这些特征对当前主流的面向对象方法形成新的挑战,面向 Agent 方法应运而生.由于面向 Agent 方法提供了诸如 Agent、协同等高层的抽象概念,能够对具有复杂行为特征的实体和系统进行自然建模问题,因此,人们有理由相

* Received 2005-06-27; Accepted 2006-04-03

信它能够解决当前具有诸如自主、主动适应、动态可伸缩、开放可成长等特征的复杂应用问题,因而,其研究和应用引起了学术界和工业界的关注和重视.

近些年来,相关研究的确取得了许多进展,也提出了一些有代表性的面向 Agent 软件开发方法[1−4],但相关的应用状况却并不如人们所预期的成功.这主要表现在当前绝大多数分布应用的开发采用的仍是面向对象方法而不是面向 Agent 方法.究其原因:一方面是因为面向 Agent 方法的发展历史尚短,虽然上述有代表性的工作在系统分析与设计阶段的建模方面已取得了较大进展,但系统实现阶段的支持工具(即面向 Agent 程序设计语言及其支撑环境)远未成熟;另一方面,面向对象方法之所以得到广泛采用还因其不仅有成熟的软件开发方法,还有成熟的软件工程全过程的支持工具.所以,面向 Agent 方法尚未得到成功与普及应用的主要问题是尚缺乏成熟的面向 Agent 程序设计语言及其支撑环境.

从面向对象方法的发展历史类比来看,上述主要问题的解决应该还需要一个比较长的过程.何况,主要源于人工智能领域的、作为面向 Agent 方法建模基本构件的 Agent 在概念上尚未达成共识,特别是对 Agent 在软件实现层面上应该是怎样的软件实体这一基础问题并未形成一致认识,而这对于上述主要问题的解决至关重要.因此,我们提出:当前面向 Agent 程序设计语言及其支撑环境研发的关键问题之一是“Agent 应该怎样软件实现”.在研发相应的语言及其支撑环境时,着重考虑了这个关键问题的研究.本文将总结我们关于此问题的研究成果,旨在推进对此问题的深入认识与解决.

1 相关工作

当前,面向 Agent 方法的研究热点主要集中在多 Agent 系统上,与前言所提出的关键问题相关的研究工作主要还集中在较早期有关单 Agent 的理论与体系结构上.例如,Bratman[5],Rao 与 Georgeff[6]的 BDI 结构的实用推理 Agent;Brooks[7]的归类式(subsumption)结构的反应式 Agent 以及 InteRRaP[8]混合式结构 Agent.由于这些工作的重点在体系结构上,虽然也提到一些实现研究,但并没有就相应的实现支持工具做深入研究.在面向 Agent 程序设计语言方面比较有代表性与广泛影响的工作是 Shoham 提出的 Agent0 语言[9].文献[9]主要论述基于思维意识立场进行程序设计的可行性与方法,主要内容是将程序设计思想用信念、愿望以及意图等思维属性表示出来.但 Agent0 语言还只是一个语言原型,没有形成相应的实现支持工具.另一方面,现已出现一些有一定影响力的商业化面向 Agent 软件开发环境(诸如 AgentBuilder,Aglet,JACK 等),但出于商业化考虑,它们没有公开怎样软件实现 Agent,有关它们的实现方法的评价研究更未展开.因而,Agent 的特征实现得怎样仍是个问题.

所以,前言中提出的基础性关键问题仍值得深入研究.我们在对上述从体系结构到商业化产品等相关工作的调研与借鉴以及进一步研究的基础上,提出该关键问题的当前可行的解决方案是在现有成熟的面向对象方法的基础上进行面向 Agent 扩展,但关键是怎样扩展才能使得用扩展后的方法实现的 Agent 的确具有 Agent 特征?这就需要从描述 Agent 特征的 Agent 概念以及它与对象概念的区别与比较入手展开分析.

2 对象和 **Agent** 的对比分析

关于 Agent 概念本身,由于对 Agent 各种特征的理解与重视程度在不同的领域有所不同,迄今为止,尚没有一个一致认可的 Agent 定义.本文主要关心 Agent 应该怎样软件实现的问题,因此,我们采用这样一个所谓 Agent 的弱定义(在文献[10]的 Agent 定义上稍有改动):驻留于某一环境中,具有灵活的自主性(包括反应性、主动性和社会性)行为特征的计算机系统.这个弱定义一方面用能体现人工智能的外部行为特征(它们也是人们有了对象概念后仍对 Agent 感兴趣的根本原因)来描述 Agent,另一方面也指明 Agent 是驻留于某一环境中的计算机系统.本文将采用这个弱定义,并将其中的计算机系统限定为软件系统,称其为软件 Agent.应当注意:该定义仅是从最高抽象级别给出软件 Agent 的需求定义,而具体怎样设计与实现它则正是本文的关键问题.至于所谓 Agent 的强定义则主要是指在弱定义的基础上进一步要求 Agent 具有学习与主动适应等更强的行为能力,并不必一定限制为计算机系统.

关于 Agent 与对象的区别,我们首先注意到,虽然对象与 Agent 都可作为概念建模的基本构件,是对问题领

域中实体的抽象描述,但它们的起源背景却有根本区别:对象概念源于计算求解领域,而 Agent 概念源于问题领域.因此,无论对象及其属性和服务在用于概念建模时可以如何抽象,其在软件实现层面上却非常清楚地是数据与方法的封装实体.而对 Agent 来说,这一点尚无定论,还需要深入研究.

通过从 Agent 与对象两个概念本身、面向 Agent 与面向对象两种方法的基本观点和技术手段等方面作进一步分析比较,可见两者的主要区别如下:

- 对象通过属性和服务的结合的确能够表述事物的静态特征和动态特征,但其动态特征的展现是被动的,因而不具有灵活的自主性行为.Agent 则特别强调灵活的自主性行为,这也是 Agent 在概念上与对象的根本区别;
- 面向对象方法将事物发展变化的起因归结为对象之上更高层面的事情,而没有在其建模的基本构件(对象)中加以考虑.面向 Agent 方法则采取事物、人和环境相结合的观点来认识客观世界,它将事物发展变化起因的抽象也纳入新的建模基本构件 Agent 之中,以反映事物动态特征展现的本质与根源,从而使其具有能处理好文章开始部分中所述的复杂意愿性特征的基础.因而,面向 Agent 方法更贴近当前许多分布求解新需求的真实情况;
- 面向对象方法通过封装技术反映事物的相对独立性,也增强软件的可维护性,通过类与继承技术反映事物组成的结构层次,也获得软件的可重用性.这些技术在面向 Agent 方法中不仅继续保留而且还需进一步发展,保证 Agent 灵活自主性的根本是对其心智状态和心智活动的合适封装,它比对象的数据与方法的封装复杂,其原因在于心智状态的定义方式及维护机制和心智活动中有关方法的使用方法等.

所以,Agent 与对象两者的确具有本质上的区别:单纯标准的对象无法体现 Agent 概念中描述的灵活的自主性特征.但通过上述比较还可见这样的可能性,即可将软件 Agent 视为特殊的基于对象的软件系统,它是通过利用现有成熟的面向对象方法进行旨在弥补上述区别的面向 Agent 扩展后的方法开发而成,从而使其的确具有 Agent 弱定义中描述的灵活的自主性特征.下面两节讨论按此技术途径展开研究的一些关键技术细节.

3 软件 Agent 的实现体系结构及其内部行为自主决策算法

在已基本明确软件 Agent 这种特殊的基于对象的软件系统的开发需求后,接下来就是设计该系统的实现体系结构(注:这里引入所谓实现体系结构,意指它比通常的 Agent 体系结构带有更多的旨在开发能弥补第 2 节所述区别的软件 Agent 的实现支持工具的特定需求).Agent 体系结构作为构造 Agent 的特殊方法学,它定义和描述了组成 Agent 的基本成分及其作用、各成分之间的联系和交互机制以及 Agent 的内部推理机制.由于对 Agent 概念的深入理解和认识与 Agent 体系结构设计直接相关,从概念出发对 Agent 体系结构进行研究已有许多,例如第 1 节中已提到的一些著名的体系结构.由于体系结构是实现体系结构的基础,下面,将 BDI 体系结构着手进行操作.

3.1 Agent的BDI概念模型

Bratman[5]所提出的 BDI Agent 概念模型如图 1 所示,它由 7 个基本成分组成,分别是信念库、愿望库与意图库以及信念维护器、愿望产生器、意图产生器与动作执行器这 4 个功能部件.文献[5]中,这些基本成分之间的关系非常复杂,但它们能够比较全面地反映模仿人类推理的实用推理过程.

Rao 和 Georgeff[6]进一步给出了形式描述该概念模型的 BDI 逻辑,论述了意图在实用推理中的重要性以及它在逻辑上并不能归结为信念和愿望,并且指明:在具体应用时,应针对实际需求舍弃 BDI 框架的部分描述能力并作出一定的简化改进.对此,我们认为,由该概念模型能构建符合前述强定义的 Agent.但面对本文的软件 Agent 需求,如 Rao 和 Georgeff 的建议,则需要对它作进一步简化改进.

3.2 软件Agent的实现体系结构设计

将软件 Agent 构造为基于对象的软件系统的关键在于:应该怎样对现有的成熟的面向对象方法进行适当的扩展,然后用来进行该系统的设计与实现,以期望能在系统层面达成对心智状态和心智活动的合适封装,从而使

该系统能够反映事物发展的起因,体现灵活的自主性特征.对此需求,我们在参考 Rao 与 Georgeff 在文献[6]中提出并成功应用于 PRS 与 dMARS 两个著名推理系统中,以及对 BDI 概念模型的简化改进思路的基础上,考虑了相应的简化改进,并提出如图 2 所示的软件 Agent 实现体系结构.

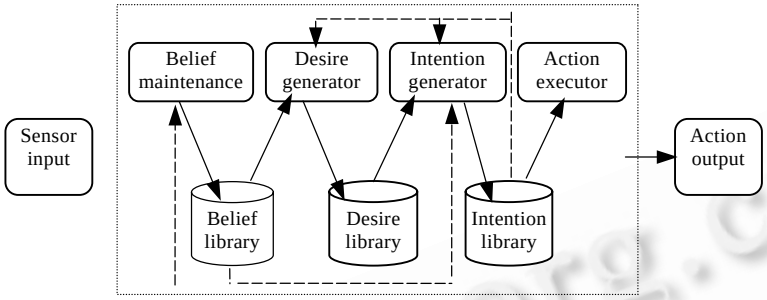

Fig.1 BDI conceptual model of Agent

图 1 Agent 的 BDI 概念模型

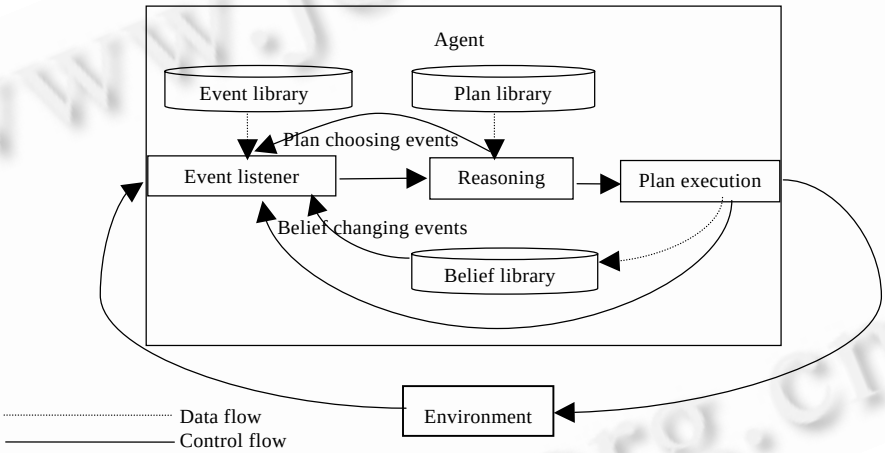

Fig.2 Event driven architecture of software Agent

图 2 基于事件驱动的软件 Agent 实现体系结构

如图 2 所示的软件 Agent 实现体系结构对如图 1 所示的 BDI 概念模型的主要简化改进是:

(1) 从 Agent 心智活动的起因来看,BDI 概念模型中,Agent 对外界环境的感知与其信念的比较是其心智活动的起因.这的确模拟了人的心智活动的起因,但它同时也带来了复杂性,特别是与 Agent 自身愿望和意图有关的维护是更高层面的心智活动.在当前软件 Agent 的需求下,应当假设软件 Agent 的计算资源与能力均是有限的,其被指派代理实现的愿望(目标)应该不太复杂,上述更高层面的心智活动可不予考虑,它不能也不必对所处环境中任何变化都作出反应,而只需对与其被指派目标相关的事件进行感知.因此,图 2 结构简化为以目标事件驱动作为起因.

(2) 在(1)中对起因简化后,应对 BDI 概念模型中描述心智状态的 3 个知识库进行表 1 第 2 行所示的实现层次上的简化改进.

Table 1 Corresponding representation of the BDI concepts in two levels

表 **1** BDI 概念在 2 个层次上的对应表示

Philosophy level	Belief library	Desire library	Intention library
Implementation level	Belief library	Event library	Plan library

其中,对比哲学层次上的信念库,在实现层次上的信念库简化改进为主要描述软件 Agent 感兴趣的外界信息.此外,愿望库与意图库也作相应改变:将愿望库转化为具体的软件 Agent 感兴趣的事件库,这样做,实际上是将可选目标间接地归结到引发它的事件起因上;至于意图,在实现层次上,它将间接地归结到正在执行的或将要执行的计划,因此,意图库转化为具体的相应事件处理的计划库.

(3) 在(1)中对起因简化后,还应对 BDI 概念模型中的 Agent 心智活动进行简化.图 2 中已没有对更高层面的有关 Agent 自身理性平衡的心智活动进行直接考虑.这样做,无疑将限制 Agent 强定义中描述的学习与主动适应等能力,但重要的是:这样,在其改进了 BDI 概念模型追求推理逻辑上的完备的同时,却可能不具有可计算性的特征[6].此时,图 2 中相关的心智活动已相对简化为基于事件驱动的实用推理过程,即根据给定的信念库、事件库及计划库进行相应的"事件-实用推理-计划执行"的过程,从而具有可计算性.该过程具体情况将在第 3.3 节中加以阐述.

应当强调:上述起因、可计算性方面的改进带来的主要益处是,图 2 可用适当扩展后的面向对象方法加以实现.

3.3 基于事件处理的软件Agent内部行为自主决策算法

第 3.2 节所述的基于事件驱动的实用推理过程是相对简化的,主要含义是指其具有可计算性,因此,也称该过程为基于事件处理的软件 Agent 内部行为自主决策算法.实际上,该过程并不简单,图 3 示意了该过程.

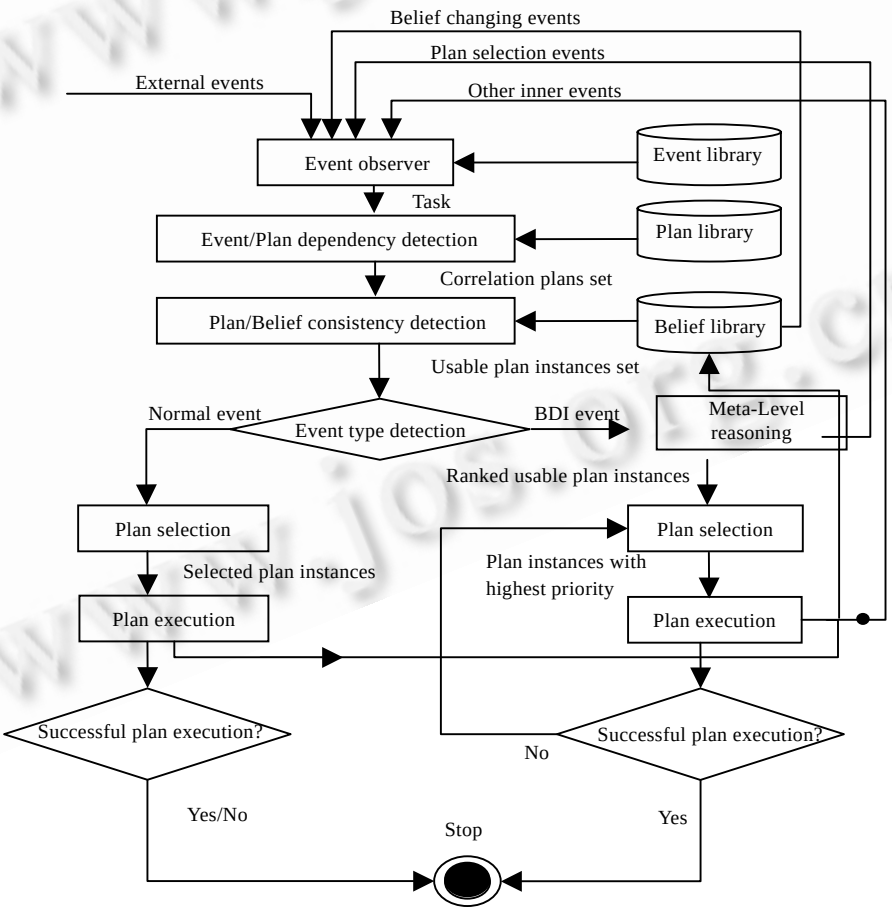

Fig.3 Behavior decision process of event driven software Agent

图 3 基于事件驱动的软件 Agent 内部行为决策过程

如图 3 所示,当软件 Agent 通过事件观察器获知某一事件发生后,它先要查看事件库,只有在事件库中登记了的事件才会被做进一步处理.这些事件包括前述与其被指派目标相关的、使其感兴趣的外部事件以及其他一些需要自身进行处理的事件(如信念改变事件、推理过程中可能出现的计划选择事件)以及一些其他内部事件(如时钟事件)等.当软件 Agent 确定要做进一步处理后,它将会初始化一个任务以对该事件进行处理.一个软件 Agent 可以同时处理多个任务.针对各个任务,软件 Agent 将根据显示给定的计划库进行“事件/计划相关性检测”以得到一组相关计划集,进而根据显示给定的信念库进行“计划/信念一致性检测”,在从前所得到的相关计划集中再挑选出与软件 Agent 当前信念相符的计划,并生成相应的计划实例集.

接下来,软件 Agent 将根据普通事件和 BDI 事件两种不同事件类型采取不同的事件处理策略.对于普通事件,软件 Agent 直接从上一步所得到的计划实例集中随机选出一个计划实例来执行.即使该计划执行失败,也不再尝试计划实例集中的其他计划,而是结束对该事件的处理.这是因为普通事件通常表示一些“稍纵即逝”的情况,所以在计划执行失败的时候,导致对该普通事件进行处理的前提条件可能已经不存在了.软件 Agent 对普通事件这样的处理使其具有反应性特征.

与普通事件不同,BDI 事件隐式地对应一个软件 Agent 应该坚持的目标.因而,它对 BDI 事件的处理是一个目标引导的处理过程,它还需要进一步体现理性的计划选择和计划失败处理机制.这里,我们采用元级推理(meta-level reasoning)技术.所谓元级推理是指关于推理的推理.用在此处,主要是指对计划选择这一特殊事件的处理.针对不同类型的 BDI 事件,元级推理可以有不同的选择机制,它可以产生进一步的计划选择事件以在一组可用计划实例有多个可能入口选择时进行进一步的元级推理.BDI 事件主要由计划中的推理方法引起,但这些推理方法并不能由标准的面向对象程序设计方法直接提供,而是要进行一些相应的面向 Agent 扩展.这种处理方法将使得显示给定的计划库留下动态扩展的余地.从而使得软件 Agent 具有更为丰富的理性决策能力.当软件 Agent 经过元级推理获得一个按优先级排序的可用计划实例集后,在“计划选择”的后继活动中将挑选出优先级最高的计划实例来执行.与普通事件不同,软件 Agent 在处理 BDI 事件的计划执行失败时,会从按优先级排序的可用计划实例集中删除失败的计划,并选择当前计划集中次高优先级的计划实例来继续执行,以体现软件 Agent 对所追求目标的坚持.只有在按优先级排序的可用计划实例集中所有计划都执行失败的情况下,软件 Agent 才可能放弃对该 BDI 事件的处理.这种计划执行失败处理机制使得软件 Agent 可以避免那些简单推理模型的缺陷.总之,上述理性的计划选择和计划失败处理机制使得软件 Agent 具有主动性特征.

无论是普通事件还是 BDI 事件,在计划执行过程中,信念库都有可能会得到修改,也有可能产生新的内部事件,还有可能是给其他软件 Agent 或环境中其他系统发送一个消息.信念库的改变又可能会产生一个信念改变事件.

综合上述有关如图 2 所示的软件 Agent 实现体系结构与如图 3 所示相关的内部行为自主决策过程的讨论可见:通过显式给定间接表述 BDI 概念的事件库、计划库以及信念库,回避了对愿望和意图的直接维护,从而使得该设计具有可计算性基础.另一方面,通过对普通事件和 BDI 事件设计不同的处理机制,使得所设计的软件 Agent 既具有反应性又具有主动性,软件 Agent 之间可以通过相互发送消息事件方式进行交互,也使其具有一定程度的社会性,特别是设计元级推理和计划失败处理机制,使得软件 Agent 具有间接完成愿望选择与意图坚持的理性决策能力,从而真正体现 Agent 弱定义描述的灵活的自主性特征.当然,这还需要对事件库、计划库以及信念库设计较复杂的关联以及相关的扩展才能实现.

4 软件 Agent 的设计框架

本节将利用面向模式的分析与设计方法[11](pattern-oriented analysis and design,简称 POAD)进一步开发出软件 Agent 的详细设计,即软件 Agent 的设计框架[11].

POAD 方法是一种基于模式重用的系统开发方法.基于该方法开发出的所谓设计框架,是指一组设计类的集合,这些设计类对于特定领域来说是普遍适用的,设计类提供了在具体系统设计时对设计框架进行扩展和定制所必须满足的基本机制.由于本文研究 Agent 应该怎样软件实现问题,并不是要开发一个具体的软件 Agent,

而是要利用将现有成熟的面向对象方法进行适当扩展后的方法开发出具有灵活自主性特征的软件 Agent 的一般实现体系结构与设计框架,因此,选用 POAD 方法来开发软件 Agent 的设计框架是合适的.POAD 方法包括分析与设计模式选择、高层设计和设计精化 3 个阶段,下面各小节将分别给出这 3 个阶段相关的结果.

4.1 软件Agent的设计模式选择

第 3 节的讨论也可视为 POAD 方法对软件 Agent 的系统分析过程,它给出的如图 2、图 3 所示的结果已对软件 Agent 进行了功能分解与结构设计,POAD 方法接下来的工作就是为各功能结构模块选择合适的设计模式.基于第 3 节的结果以及在模式库[12]中进行检索和匹配,我们作出下列选择:首先,选择 Singleton[12]模式设计软件 Agent 的管理者,因为 Singleton 模式中只有一个类,即所谓单例类,它只能有一个实例且必须由自己创建并为所有其他对象提供这个实例,这样正好符合软件 Agent 的管理者的需求,即软件 Agent 中只有一个管理者,保证软件 Agent 自己创建自己,并且其他对象都可以访问到它;其次,选择 Mediator[12]模式来设计图 3 中事件观察器,因为 Mediator 模式由一个中介对象来封装一系列对象的交互,中介者使各对象不需要显式地相互引用,这样正好符合事件观察器的功能需求;再次,选择 Abstract Factory[12]模式可以设计实现图 3 中“可用计划实例集”部分的功能;最后,选择 Strategy[12]模式可以设计实现图 3 中“软件 Agent 对不同的事件类型采取不同的事件处理策略”部分的功能.

4.2 软件Agent的模式级设计模型

POAD 方法的高层设计阶段是将第 4.1 节选择的各设计模式进行合成,以得到模式级设计模型,它主要包括设计模式的实例化,并确定模式实例之间的关系.

所谓模式实例化,即是对所选择的一般设计模式指定一个适合具体应用系统的实例名.就软件 Agent 而言,我们指定软件 Agent 的唯一管理者的 Singleton 设计模式使用 AgentManager 作为模式实例名,而其他 3 个模式依次实例化为 EventObserver,ApplicabePlansProducer 以及 EventHandlingStrategy.

至于这些模式实例之间的关系,因为模式的选择是基于如图 3 所示的过程进行的,所以,这些模式实例的依赖关系比较明确:AgentManager 创建后,将使用 EventObserver 进行事件的感知;一旦感知到一个需要处理的事件后,它将使用 ApplicabePlansProducer 来生成可用计划实例集;接着,ApplicabePlansProducer 又要使用 EventHandlingStrategy 来对不同类型的事件采取不同事件处理策略;在 EventHandlingStrategy 对事件进行处理的过程中,还可能对 EventObserver 产生一个新的事件.

在 POAD 方法中,要将上述概念中的依赖关系进一步映射为较低层的模式接口之间的关系.首先,要为所有模式实例声明模式接口[11].模式接口既可以是接口方法也可以是接口类.在文献[11]中给出了一些著名设计模式的接口.其中,Singleton 模式的接口是方法 *getInstance*().通过该接口方法,系统中其他对象就可以知道自己的“管理者”是谁.Mediator 模式有两个接口:一个是接口类 Colleague,事件源通过该接口类而能够向中介者报告有事件需要处理;另一个是接口类 Mediator,中介者通过该接口来让具体的同事对它所收到的事件处理请求进行具体处理.AbstractFactory 模式有两个接口:一个是接口类 AbstractFactory,它为创建不同类型的产品提供了接口;另一个是接口类 AbstractProduct,它为工厂创建的所有产品提供了接口.Strategy 模式有两个接口:一个是 Strategy 接口类;另一个是 Context 接口类.使用 UML 类别模板将上述讨论总结为软件 Agent 的模式级设计模型,如图 4 所示.

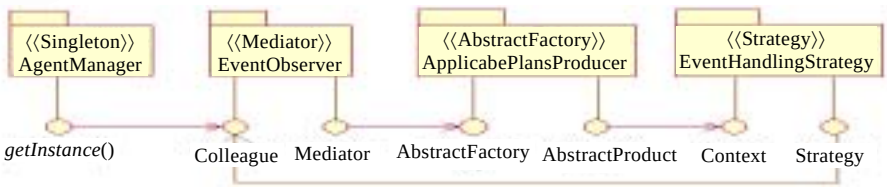

Fig.4 Pattern-Level design model of software Agent

图 4 软件 Agent 的模式级设计模型

4.3 软件Agent的设计框架

POAD 方法的设计精化阶段将对模式的内部设计添加系统的具体信息,进行模式内部细节的实例化,并对模式实例的内部组成进行删减、合并等设计优化,最终形成优化的设计框架.

首先,对 Singleton 模式的实例 AgentManager 进行内部细节的实例化.将该模式中唯一的一个类命名为 Agent.按前述软件 Agent 的管理者唯一、自己创建自己的实例以及向所有其他对象提供这个实例的功能需求,类 Agent 的构造子 *Agent*()设为私有,且再通过声明一个私有静态变量 *AgentXXX*:*Agent=new Agent*(),可实现前两个需求.将 AgentManager 中的接口方法 *getInstance*()具体命名为 *getAgentName*(),它作为一个公共方法,通过它可以实现第 3 个需求.另外,每个 Agent 都有自己的信念、所能处理的事件以及相应的计划.它们可用私有变量 *myBelief*、数组 *myEvents*[*m*]、数组 *myPlans*[*m*][*n*]来分别表示.

接着,对 Mediator 模式的实例 EventObserver 进行改进和内部细节的实例化.由于每个软件 Agent 中只有一个事件观察器,该角色可以由软件 Agent 自身扮演,所以将 EventObserver 与 AgentManager 进行合并,EventObserver 中的具体中介者 ConcreMediator 类与 AgentManager 中的 Agent 类合并,仍命名为 Agent,它既是软件 Agent 的管理者,又是事件观察器与中介者.ConcreMediator 类中的公共方法 *ColleagueChanged*()将作为合并后 Agent 类的方法而命名为 *pleaseHandle*(*EventType*,*EventName*),用它向相应的对象发出相应的事件处理通知.至于 EventObserver 中的接口类 Colleague,由于其部分功能可由 Agent 类的方法 *getAgentName*()实现,而且软件 Agent 也不再与事件源发生直接关系,所以,可将 Colleague 类删除.需要强调的是:*pleaseHandle*()方法进行事件发送前,将先对事件源所传来的事件名 EventName 进行判断,如果 EventName 在数组 *myEvents*[*m*]中,它将如下段所述继续处理;否则,它将放弃对该事件的处理.

随后,对 Abstract Factory 模式的实例 ApplicabePlansProducer 进行改进和内部细节实例化.因为 Abstract Factory 模式试图用一个工厂族负责产生多个产品族,而软件 Agent 需求的产品族只有一个,就是“可用计划实例集”,所以,可将该模式改造为用一个工厂族负责产生一个产品族.具体改造为:首先,将其唯一的抽象产品类命名为 Plan,而抽象工厂类 AbstractFactory 中的产品生成方法只保留一个,并将该方法命名为 *productApplicablePlans*(*EventType*,*EventName*)方法,它将成为模式实例的接口.当软件 Agent 通过 *pleaseHandle*()方法判断需要继续处理并发出相应的事件处理通知后,它将产生一个新的线程,并在新的线程中创建相应的以 EventName+Factory 形式命名的具体事件工厂类,而后,调用该类的 *productApplicablePlans*()接口方法以产生计划实例集.在 *productApplicablePlans*()方法的执行过程中,首先要对 Agent 中声明的相关计划依次调用相应计划中的 *isRelevant*(*EventName*)静态方法,只有返回值为真的计划才是与事件 EventName 相关的.然后,还要调用相应计划中的 *isApplicable*(*EventName*)静态方法,只有返回值为真的计划才是与软件 Agent 当前信念一致的.至此,才会真正创建计划实例,并将所创建的计划实例放入一个数组 *AppPlans*[]中,供下一阶段继续处理.静态方法 *isRelevant*()与 *isRelevant*()在每个计划中都要定义,它实现图 3 中“事件/计划相关性检测”与“计划/信念一致性检测”两个计划过滤机制.在抽象类 Plan 中还定义了一个 *body*()方法,计划的具体执行步骤都可在该方法中定义.

最后,对 Strategy 模式的实例 EventHandlingStrategy 进行改进和内部细节实例化.该实例中的接口类 Context 持有一个对另一接口类 Strategy 的引用,Context 类中有一个公共方法 *contextInterface*(),当 *productApplicablePlans*()方法在选出一组可用计划集 *AppPlans*[]后,将调用这里的 *contextInterface*()方法,并传入参数 *AppPlans*[].如图 3 所示,软件 Agent 对普通事件和 BDI 事件的不同处理策略,抽象的 Strategy 类主要有两种具体的策略类 NormalHandleStrategy 和 BDIHandleStrategy.有必要指出:具体使用哪种策略并不是由 EventHandlingStrategy 来决定,而是 *productApplicablePlans*()根据事件的类型在通过接口类 Context 间接引用接口类 Strategy 时作出决定.

经过上述 4 段讨论,也就完成了 POAD 方法重用已有优秀设计模式及其内部组成的主要过程,并将各模式实例之间的接口关系进一步映射为更低层的设计类之间的关系.为了得到软件 Agent 最终的设计框架,还需要去掉模式外衣并进行适当的遗漏弥补与优化精简.就软件 Agent 而言,在前述设计中,Agent 类通过变量 myBelief、数组 *myEvents*[*m*]引用了对应图 2 事件库与信念库的 Event 类与 Belief 类,于是,需要弥补添加 Belief

类与 Event 类、相关属性与方法以及它们与其他类的关系等.Belief 类中应该包括一系列的关键字段和相应的值字段,这些信念值字段可被特定的用户读写,还可能产生 BeliefChange 事件.至于 Event 类,其中应该包括事件类型、事件的触发条件、事件发生时可能附带的消息等.另外,出于简化考虑,具体事件工厂类与具体计划类只取 EventAFactory,ConcretePlan1 作代表.这样,最终得到了如图 5 所示的软件 Agent 的设计框架.

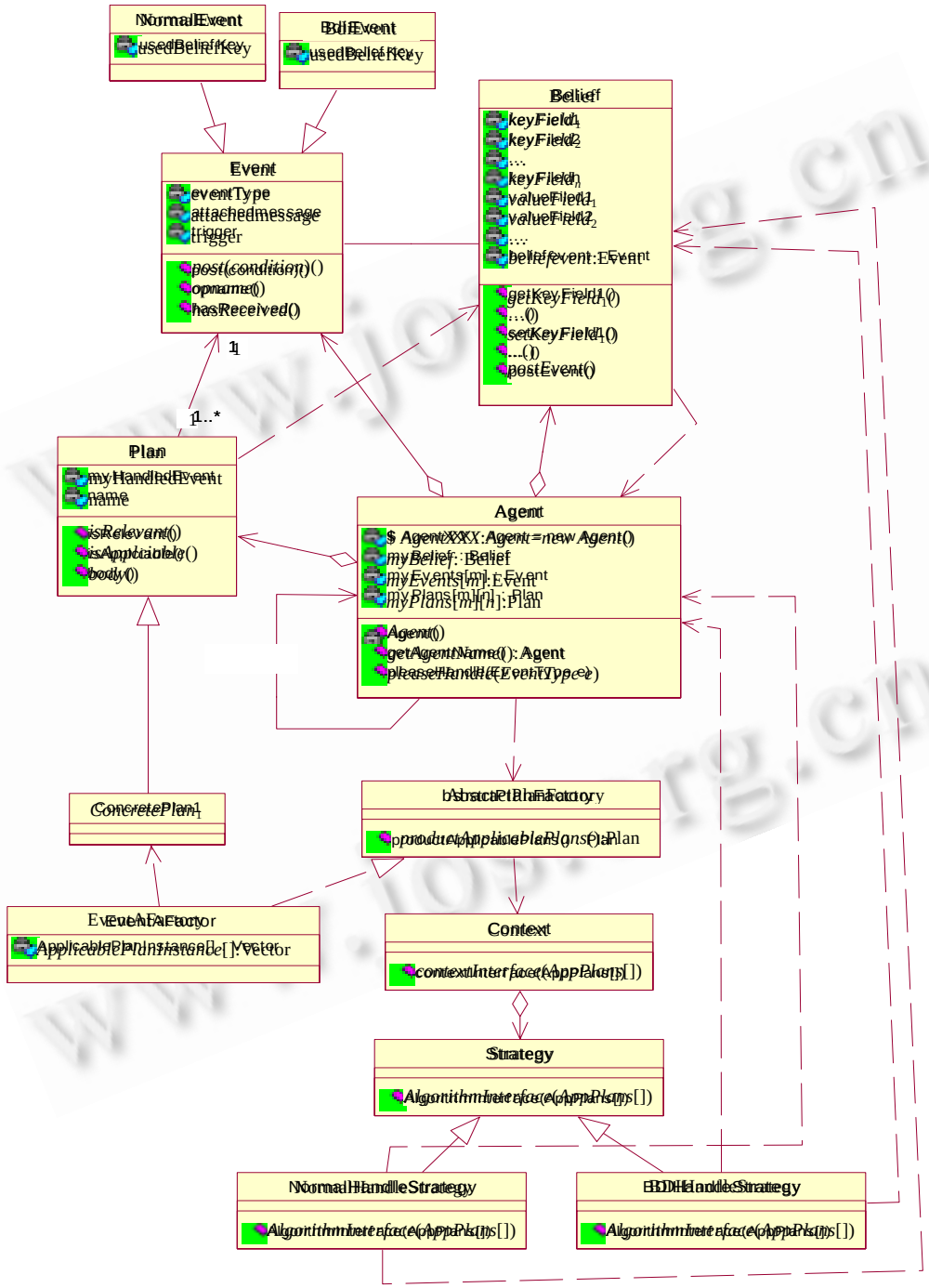

Fig.5 Design framework of software Agent

图 5 软件 Agent 的设计框架

从图 5 可以看出,软件 Agent 的设计框架还是比较复杂的.系统中的各个类,尤其是代表软件 Agent 意识属性的 Belief,Event,Plan 类之间,通过 Agent 类存在耦合度非常高的复杂关联关系,而且,这些关系通过图 5 中描述的推理过程还得到了深化.

5 结束语

通过第 3、4 节软件 Agent 的实现体系结构与设计框架,可对本文前言提出的关键问题,以及怎样用对现有成熟的面向对象方法作适当扩展后的方法开发出符合 Agent 弱定义的软件 Agent 得出比较深入的认识.软件 Agent 不可能由某一个类的实例来实现,而是应由多个复杂关联的扩展类的实例构成的系统来实现.如果希望在现有成熟的面向对象方法基础上开发出软件 Agent 程序设计语言及其支撑环境,则需要在类的属性与方法以及类本身等层次上作出扩展,以表述该设计框架所展现的类之间的复杂关联,从扩展后语言及其实现机制上保证由其所开发出的软件 Agent 的确具有灵活的自主性特征.我们还进一步开发了相应的软件——Agent 程序设计语言与支撑环境的原型系统,从某种程度上验证了上述设计思想的有效性.但还应指出:本文重点在研究单个 Agent 的构造与实现上,因而在社会性方面考虑较少,尚不完全符合多 Agent 系统在社会性方面的需求.另外,Agent 强定义中的学习与主动适应能力的设计与实现也未作考虑.这些都有待进一步研究.

References:

[1] Wooldridge M, Jennings NR, Kinny D. The gaia methodology for Agent-oriented analysis and design. Int'l Journal of Autonomous Agents and Multi-Agent System, 2000,3(3):285−312.

[2] Bauer B, Muller J, Odell J. Agent UML: A formalism for specifying multi-Agent software systems. Int'l Journal of Software Engineering and Knowledge Engineering, 2001,11(3):207−230.

[3] Wooldridge MJ, Weifl G, Ciancarini P. Agent-Oriented software engineering II. LNCS 2222, Springer-Verlag, 2001.

[4] Giunchiglia F, Mylopoulos J, Perini A. The tropos software development methodology: Processes, models and diagrams. In: Giunchiglia F, Odell J, Weiß G, eds. Proc. of the 3rd Int'l Workshop on Agent-Oriented Software Engineering (AOSE 2002). Bologna: Springer-Verlag. 2002. 162−173.

[5] Bratman ME, Israel DJ, Pollack ME. Plans and resource-bounded practical reasoning. Computational Intelligence, 1988,4: 349−355.

[6] Rao AS, Georgeff M. BDI Agents: From theory to practice. In: Georgeff M, ed. Proc. of the 1st Int'l Conf. on Multi-Agent Systems (ICMAS'95). San Francisco: ACM Press, 1995. 312−319.

[7] Brooks RA. Intelligence without representation. Artificial intelligence, 1991,47:139−159.

[8] Muller J. A cooperation model for autonomous agents. LNAI 1193. Berlin: Springer-Verlag, 1997. 245−260.

[9] Shoham Y. Agent-Oriented programming. Artificial intelligence, 1993,60(1):51−92.

[10] Wooldridge MJ. An Introduction to MultiAgent Systems. John Wiley & Sons, Inc., 2002.

[11] Yacoub S, AmmarR H. Pattern-Oriented Analysis and Design: Composing Patterns to Design Software Systems. Addison-Wesley, 2003.

[12] Gamma R, Johnson HR, Vlissides J. Design Patterns-Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

黎建兴(1962－),男,湖南长沙人,副教授,主要研究领域为面向 Agent 程序设计语言,面向 Agent 需求分析.

束尧(1976－),男,博士生,主要研究领域为高性能评价,软件工程,Agent 技术.

毛新军(1970－),男,博士,教授,CCF 高级会员,主要研究领域为软件工程,Agent 理论和技术,分布计算.