计算机科学 1998 Vol. 25 №. 3

面向 Agent 软件工程(Ⅰ):Agent 计算、编程风范与语言设计

Agent-oriented Software Engineering(Ⅰ):Agent-oriented Computing, Programming and the Design of Agent Programming Language

樊晓聪 徐殿祥 侯建民 陈冠岭 郑国梁

(南京大学计算机科学技术系 计算机软件新技术国家重点实验室 南京 210093)

摘 要 This paper introduces the characteristics of agent-oriented computing, agent-oriented programming and the design of agent-oriented programming language. From the pointview of software engineering, their main problems existed are analized, solutions to these problems are provided, and future works are also discussed.

关键词 Agent-oriented software engineering, Agent-oriented computing, Agent-oriented programming, Agent-oriented programming language.

1 引言

Agent 技术源于人工智能,现已融入主流计算机的各个领域[1]。面向 agent 的风范[2]自提出以来一直受到广泛关注,各种类型的软件 agent 大量用于信息处理、办公自动化、交通管理、私人助手等领域,特别是 Internet 信息网络的飞速发展,以及 Internet 本身固有的开放、复杂、异质、动态、分布等特性,为 agent 技术的充分发展带来了有利条件和有力挑战,开创了面向 agent(AOC)的新时代[3]。但目前对 AOP 在理论上的探入研究还不多见,这大大限制了 AOP 的良性发展。

为使 AOP 走向成熟并渐趋实用,我们必须从软件工程的角度,分析面向 agent 计算的优点和局限,研究怎样才能扬长避短,充分发挥 AOC 的潜力。此外,AOP 究竟有什么优点,为什么要提出 AOP,为什么现有的技术(如 OOP)不能满足需要?新的编程风范提出之后,首先要解决的问题就是编程语言的设计。那么,我们如何来设计 agent 编程语言呢?在设计时应当注意哪些问题,应当从传统语言设计中吸取哪些经验?

软件工程关注软件的质量。因此,另一个需要解决的问题是,如何规范、实现和验证面向 agent 的系统。

本文主要从软件工程的角度介绍面向 agent 计算、编程风范以及 agent 语言设计的特点,分析它们存在的主要问题和解决途径,并讨论不同问题的未来研究方向。文[4]讨论和处理如何规范、实现和验证面向 agent 的系统。

2 面向 agent 计算(AOC)

2.1 AOC 的优越性

面向 agent 的计算被誉为"软件开发的又一重大突破"[5]、"软件界的新革命"[6]。英国一项市场调查表明,agent 软件将从 1994 年的 $ 3700 万增长到 2000 年 $ 26 亿。任何一项新技术要想占领如此大的计算机市场,至少应具备下列条件之一:1)能够解决迄今为止其它方法不能涉足的问题——要么还没有技术能够解决这类问题(很少),要么采用现有方法解决过于昂贵;2)能够以一种更好(更自然、更容易、速度更快、效率更高)的方法解决现有问题。相对于第一个条件,面向 agent 的系统正致力于解决而传统方法不能处理的问题有:

1)开放性:如像高度开放的软件环境 Internet,是一个松散耦合的计算机网络,伴随着规模与复杂性的日益扩张,设计开发能够充分利用 Internet 巨大潜力的软件工具及相关技术是 1990 年代计算机科学家们面临的最有力挑战。在 Internet 这样的环境中,基于 agent 方法之所以如此重要,在于用户预先不能确知与其交互的各种资源。这要求 agent 以

及它们之间的接口必须具有灵活性和坚定性。但是，在设计 agent 时很少能预测到将来可能发生的交互行为。因此 agent 必须具有高超的社交能力，比如使他人同意自己观点的劝说能力[7]，有助于达成一致的协商能力[8]，控制协同工作的协调机制[9]等，所有这些社交能力必须依赖于底层的通信机制(以都能理解的方式传输言语行为信息)。开放性还要求 agent 能不断地监控它们所处的环境，并在适当时机主动产生新的目标。

2)复杂性：问题领域的广泛性、繁杂性和不可预测性，使得通用系统的开发是不可行的，唯一的解决途径是开发一些能够解决特定领域问题的专用模块化组件。agent 实际系统的成功经验表明，一个大的求解问题可以分解成许多较小、较简单的问题。这些专门用来解决子问题的子系统易于开发维护，并且效率高。分解策略使得每个 agent 可以采用最合适的方法解决自己特定的问题，而不必采用统一的开发模式使整个系统折衷求全。面向 agent 系统比起传统软件工程中经典的"分而治之"策略来说，其优越性在于：agent 能够以灵活的、上下文有关的方式(因此需要社交能力、应激性和主动性)，而不是通过一些预先严格确定的接口函数，与外界进行交互作用。此外，环境的不可预测性要求 agent 既能对环境变化作出反应(对出乎意料的情况能明察秋毫)，又能根据目标主动规划自己的行为。

相对于第二个条件，agent 技术具有以下优点：

1)数据、控制、专家知识或资源的分布：当问题领域涉及大量不同的问题求解实体(或数据资源)，这些实体在物理或逻辑上分布(相对于它们的数据、控制、专家知识或资源)，并且需要相互协作(或合并)以解决公共问题时，agent 技术是一种有效的选择。例如，在一个分布的保健系统中，普通实习生、医院的大夫、护士和家庭护理组织必须协同工作为病人提供周到的服务[10]。这种情况存在着数据(普通实习生和医院护士具有不同的病人数据)、控制(每人负有不同的任务和责任)、专家知识(大夫的知识不同于普通实习生和医院护士的知识)、资源(病床、药品等)的分布，最适于采用 agent 技术。面向 agent 的技术为此类问题提供了非常自然的建模方法：现实世界的实体和它们之间的关系可以直接映射成具有自治问题求解能力的 agent，它们拥有自己的资源和专家知识，以及协作求解问题的交互能力。此外，在数据资源的控制分布的情况下，采用 agent 技术使大量的数据处理在数据源处进行，只需交换少量的高层信息，减少了将大量原始数据传送到远地中央处理器的操作，提高了网络利用率。

2)描述自然：自治 agent 的表示方式简单明了，软件的功能可以从其名字的喻义上推敲出来。会议调度软件可以自然地表示成一个自治的、具有社交能力的 agent，被人格化用以反映用户的偏好，并代表用户与其它类似的 agent 交互。另外，在计算机游戏和虚拟现实系统中，角色可以自然地表示为自我包含的、自治的、具有社会性的问题求解实体(即 agent)。

3)现有软件系统的集成：在许多传统的计算(计算机集成制造)与过程控制领域，大量的现有软件(特别是信息系统)在机构中起着重要作用。为满足日益变化的商业需求，需要不断地修改软件。但是，随着时间的推移，系统结构和详细设计信息很可能受到破坏，使得修改这些现有软件非常困难；而完全重写即使可能，其开销也令人望而止步。因此，从长远的观点看，使旧系统能继续发挥作用的唯一途径，是将它们集成到一个大的协作群体中，其它软件系统可以利用旧系统所提供的特有功能。这可以通过给旧软件包装一层"agent 外壳"，使其能与其它系统交互操作[11]来实现。

agent 技术为复杂系统的开发和实现提供了新途径。许多新型的软件系统只能采用 agent 方法，而不能采用传统的软件工程模式(提供一个完整的系统规范，然后通过一些严格、确定的模块来实现)。基于 agent 系统的开发方法要求构造一些复杂的、自我包含的组件，并且使它们能与其它一些独立开发的类似组件灵活交互。交互作用不再通过严格预定义的接口进行，而是通过参与者之间的协商、劝辩、承诺、同意等行为来实现。这意味着系统的整体性质和行为不能预先在程序中固定，而是在运行时通过各参与 agent 的行为和交互作用来动态体现。

2.2 AOC 存在的问题

尽管面向 agent 技术在开发新型应用中起着重要作用，涌现出了大量的基于 agent 系统软件，但它并非万能良药。目前采用 agent 技术开发的大部分应用系统基本上都可以采用传统技术解决。一个特定问题领域的开放性以及含有旧系统并不说明 agent 技术是最佳(或可行)的解决途径，因为对系统设计方法的最终选择还依赖于大量的技术和非技术因素。

此外，agent 技术本身固有的缺陷限制了它的应用领域。第一个问题是，整个系统是不可预测、不确定的：哪个 agent 将与其它哪个 agent 以什么方式交互来实现什么目标，这是不能预先确定的。由于 agent 可以自由作出决策，我们不能保证 agent 之间的依赖关系能得到有效管理，由此可能导致莫明其妙的死锁和饿死。第二个问题是，整个系统的性质和行为在设计阶段不能确定。虽然可以给出个体 agent的行为规范，但由于整体行为只有运行时才能体现，致使设计时不能确定整个系统的行为规范。

有关 agent 系统的设计也存在许多问题。

1)用来设计 agent 和面向 agent 系统的方法学的开发：有关面向 agent 系统开发方法学的研究凤毛麟角，但是，agent 技术要想成功地走向商业化，要求系统设计者必须具有开发高性能 agent 及 agent 系统的一套结构化方法。这包括：如何才能最优设计出坚定灵活的个体 agent？如何将精心设计的组件有效地组织在一起，使整个系统具有足够的坚定性和灵活性？另外，由于整体系统的许多性质只能在运行时确定，使得系统描述相当困难。

2)评估、比较不同设计方案标准的开发：目前很少有用来系统地评估、比较不同 agent 的设计性能的标准。agent 设计者在设计时对于在什么情况下适合采用什么机制，几乎无章可循。为设计出好的 agent 系统，应当建立一系列能客观比较和评价不同方案的性能测试标准。

3)可重用工具的开发：agent 系统要想有效运行，还需要一些高层的基础设施支持，例如：通信机制和消息交换协议，对异质平台的互操作以及调试、监控能力。在大部分情况下，一个新应用必须重新实现这些底层功能。但从长远的角度来看，如果将这些底层功能抽象出来作为一套可重用工具，使设计者集中精力考虑有关 agent 特性的设计，则会大大提高软件生产率。

2.3 设计个体 agent 时应解决的问题

为使 agent 技术充分发挥其优势，在设计个体 agent 时，要妥善解决下列问题：

1)异质性：具有不同的认识领域、不同的知识表示模式、不同的问题求解风格，对自己所处世界及其它 agent 的世界观具有不同的假设，这些 agent 如何有效、公平地交互？

2)对不确定、不完备和矛盾信息的处理：由于 agent 对自己所处世界只有部分认识，并且其问题域具有开放、复杂、分布的特点，为获得一定程度的灵活性和坚定性，agent 需要能处理不确定、不完备和矛盾信息的先进机制。除了问题域的模糊性外，agent 还需要处理交互过程的不确定性(例如，agent A 会准时为我执行动作 B 吗？我能假定 agent C 会准时向我提供信息 1 吗？)，以及由此引发的消息的不确定性(例如，agent 如何处理其它 agent 送来的信息？是高度可信还是高度怀疑？如果收到的信息与自己的现有信息相互冲突，此时应如何处理？)。主要问题在于：a)异质 agent 能够将不确定信息的语义准确传送给其它 agent 的方法(例如，0 对一个 agent 来说表示"无关紧要"，对另一个 agent 可能表示"绝对不信")；b)用来对其它 agent 的行为或丢失信息进行处理的机制；c)能够找到冲突产生的根源的方法，以及消除冲突的方法。

3)实时操作：随着 agent 越来越多地被应用于实时领域，这就要求 agent 能够在时间紧迫的情况下作出及时反应。除了 agent 内部应具有调度实时行为的机制外，还需要提高自治 agent 之间社会交互的可预测性。例如，当一个 agent 认为有紧迫任务时，应能使其它 agent 优先处理这些任务。

4)适应性：由于在设计时不可能预测所有的环境变化，这要求 agent 能在运行时获取更多的有关环境和问题求解的信息。随着认识的逐渐增长，agent 应能调整行为以便更加高效地解决问题。

3 面向 agent 编程风范(AOP)

Yoav Shoham[2,12]从计算的社会性角度出发，提出了一种新型的编程风范——面向 agent 风范。其主要思想是用 agent 社会来构造计算机系统。AOP 中的 agent 从总体上说具有下列特征：1)agent 是并发执行的、自治的计算实体(进程)；2)agent 是用信念、目标等心智状态概念编制而成的认知系统；3)agent 是由描述逻辑规范的推理系统；4)agent 采用言语行为协议和消息传递的方式相互通信。面向 agent 编程风范至少有以下三个优点：

1)便于 agent 描述：使我们可以用熟悉、非技术性的语言来定义 agent。

2)嵌套式表示：使我们能够方便地表示包含其它系统描述的系统。这种嵌套表示对于必须同其它 agent 协作的 agent 非常重要。例如，agent 模型中含有其它 agent 的模型，并可能嵌套到多层。再如，在分布式系统理论中构造协议时，通常遇到这样的情

况：

如果　进程 i 知道进程 j 已经收到消息 ml

则　进程 i 应当向进程 j 发送消息 m2。

3)AOP 是一种超描述性编程风范。在过程性编程风范中，编程者需要准确指明系统应当怎么做；在描述性编程风范中，需要指明我们想实现什么，并为系统提供有关对象之间关系的一般信息，由系统内置的控制机制(如，目标定向定理证明)确定应当怎么做；在 AOP 中，我们只需给出一个非常抽象的系统规范，系统在确保其行为与内置的 agent 理论(如，著名的 Cohen-Levesque 意念模型)一致的情况下，由系统的控制机制确定应当怎么做。

AOP 风范与其它风范的主要区别在于它提供了两层抽象：自治 agent 抽象和认知 agent 抽象。

3.1 自治 agent 抽象

AOP 中的 agent 与其在 DAI 中一样，是一个自治、应激、并发执行的进程。

自治性：agent 能在没有人类和其它 agent 直接干预的情况下完成大部分问题求解任务，并对其行为和内部状态进行控制(其自身状态只能受消息传递的影响)。

并发性：agent 独立于其它 agent 而执行。

社会性：agent 应能适时地与人类和其它 agent 交互以完成自身的问题求解任务，并适时地为其它 agent 提供力所能及的帮助。这要求 agent 至少具有：向其它 agent 传递请求的通信方法，确定社交时机的内部机制。

应激性：agent 一直与其所处的环境交互，对环境中发生的相关事件能及时作出理智的反应。

主动性：agent 不只是简单地对环境变化作出反应，而是主动地产生目标并为其目标而动作。

agent 除了上述必须具有的属性外，根据不同情况，还可能具有下列性质：

适应性：根据变化的环境条件逐渐修改、完善自身的行为，增加有关问题求解的知识。

移动性：根据问题求解的需要能改变自己的物理位置。

诚实性：agent 不会蓄意传递虚假信息。

理智性：agent 为实现其目标而行动，不会无故采取阻止自身目标实现的行为。

agent 概念与基于对象并发编程风范(OBCP)中的对象概念有相似之处。在 OBCP 中，一个对象的私有状态对其它对象是隐蔽的，只能通过消息传递间接访问；对象的行为由方法或脚本确定。最大的区别在于 agent 具有"主动性"(与主动对象中的"主动"有区别)。

3.2 认知 agent 抽象

AOP 与其它编程风范的另一个明显区别是采用心智状态概念(知识、信念、期望、意图等)编制 agent。其动机是，我们在日常生活中通常采用大众心理学来解释和预测人类(复杂的智能系统)的行为。"人格化"作为一种抽象工具，为研究复杂系统提供了便利，使我们不必理解系统的实际运行机制就可以预测、解释系统的行为。

目前，计算机界正试图寻找一种好的抽象机制(过程抽象、数据类型抽象 ADTs、对象等)，我们为什么不能在计算中采用"人格化"这一抽象工具，来解释、理解并编制计算机系统呢？这就是 AOP 的主要思想。

Yoav Shoham 认为，一个完整的 AOP 系统应包含三部分：1)一个规范 agent 的心智状态和行为的逻辑系统；2)一个解释性的编程语言，编制用简单的逻辑语言规范的 agent；3)一个"agent 化"过程，从规范生成可执行的 agent。

Yoav Shoham 只发表了有关前两部分的研究成果，他采用带量词的模态逻辑(有表示信念、承诺的模态词)，其中时间采用直接引用方式。AGENT0 是一个基于规则的语言，通过规则引用 agent 的信念和承诺，对实现实际系统来说过于简单。有关"agent 化过程"的工作，我们将在[4]中介绍。

4 面向 agent 语言的设计

4.1 设计 AOP 语言的四个层次

进行语言设计时，要考虑的第一层次是程序的最终用户，第二层次是编程人员。这两类人员是判定一个程序语言是否能走向实用化的关键，如果他们不愿接受，则新型语言只能是学术殿堂中的艺术品，不可能走向市场。一种语言能否被最终用户和编程人员接受，主要在于语言的透明性、编程效率、运行效率、可移植性、灵活性、结构化程度以及底层原语的精简性等。具体到 AOPL 来说[13]，需要进一步解决：①根据真实生命的涵义来赋予并使用 agent 的心智状态属性；②逻辑推理的透明性；③对时间的处理方法；④符合社会规范与习惯。

第三层次是语言设计者，他们必须综合考虑各方面因素，以易理解、易使用为出发点来进行程序语言和语言支撑环境(编译器、解释器等)的设计。AOP

语言设计者是agent技术发展的关键性环节，面向agent语言的好坏和agent结构的优劣决定AOP的应用前景。

第四层次是理论。理论是设计程序语言和开发实现、验证工具的基础。例如，传统过程性语言的算法逻辑、算法、语法等；并发程序语言的并发理论；PROLOG的谓词演算等，而AOPL则涉及到时序逻辑和模态逻辑。

在实际进行AOP语言设计时，应当从以前的语言设计中吸取下列有益经验：

1)一个程序语言只能涉及理论基础的一些子集。例如，PROLOG只适用于HORN子句和深度优先搜索，但用户可以接受这种对谓词演算的限制。

2)编译器和解释器环境简化了语言使用的复杂性。例如高级语言中的存储管理，PROLOG中的搜索策略，AOP中的通信手段等，都可由编译器来做。

3)应在理论上研究有关编程语言的更深一层原则。例如程序的有效性验证，语言的表示能力、可判定性、易处理性等。

4.2 开放系统为AOPL设计带来的问题

AOP的最大用武之地在于开放系统。开放系统(例如，交通管理、船运公司业务管理、飞机定票、办公自动化等)对agent系统有以下需求：

存在持久性：agent应能执行自己的任务，并能随时为其它agent承担任务；

可扩展性：系统环境是动态变化的，可能有些agent会在运行时加入系统(由编程人员在程序中声明，或被其它agent激发)，也可能有些agent暂时或永远脱离系统；

分散控制：开放系统的复杂性使得集中控制不能满足实际要求；

异步消息传递：不同自治实体之间的交互需要异步通信；

冲突信息共存：通常很难维持开放系统的全局一致性，只能由agent自己进行局部一致性维护；

局域认识关系：开放系统的动态性决定了agent之间的认识关系不可能是全局的。只要当新的agent加入系统时，能在新agent与几个现有agent之间建立认识关系就可以了。

4.3 AOPL的设计

4.3.1 设计决策 目前许多传统的语言和工具可用来编制agent的"常规"行为，这要求面向agent语言能够提供一些传统语言所缺乏的支持，例如：支持位于不同计算机上的agent间通信；支持agent的自治性；使agent动作能够触发其它本地私有程序(副作用)；agent之间的交互控制和信息传递等，在设计AOPL时，需要考虑以下几个方面：

1)AOPL应包括哪些心智状态，不同心智状态之间应有什么关系。目前人们已经提出了许多用来描述agent心智状态的属性并为这些属性建立了许多可能关系。例如，有的人把愿望作为信念的子集，有的人则不赞同这种观点。事实上这应根据不同的实现目的而定。如果只是用agent模仿人类的行为，这些方法都是可行的；如果用agent技术编制计算机游戏，可以选择目标和意图；如果要编制办公agent，则最好选择任务、工作、责任来描述agent的心智状态。

2)AOPL如何处理位于不同计算机上的agent之间的通信和协作，如何选择通信原语(通常选择言语行为理论以便形成统一的标准，但也有例外)。

3)一致性处理：agent的内部一致性可以通过面向agent语言的解释器来检查。但是，如果语言描述能力太强，可能导致一致性检查无法进行甚至不可判定。在进行语言设计时应当提供一个能对大多数情况进行一致性检查的支持工具。

4)推理机制：AOP语言应为推理提供相关的编程原语和推理规程。

5)动作描述：在语言中如何表示动作。

6)从信念到动作的转化机制：agent需要根据自己的心智状态确定如何行动，因此AOP语言应为这种依赖关系提供支持。

7)agent结构的选择：应根据具体应用，选择合适的agent结构。

8)异常情况维护：可以在运行前由解释器检查出错和警告信息，或者由解释器运行时自动处理，也可以由编程人员处理。

9)继承性和创建新的agent：不同层次agent之间的依赖关系可以由编程人员手工写入程序，也可以象OOP一样来自实例化和继承机制。

10)其它：AOP应用的移植性考虑以及社会准则和社会角色等。

但是，一个语言不可能兼顾所有的情况、支持所有的特性，而只能根据最终用途来抉择。这又带来了一些问题，不同的agent可能用不同的语言、不同

的表示进行设计，同一个系统中的不同 agent 的结构，推理机制等可能各不相同，这些 agent 之间如何协作呢？这可以借鉴人类对不同语言不同习惯的处理方法：如果我想使用 UNIX 系统，我必须知道 UNIX 的操作命令；如果我想和外国人交谈，就必须要用双方都能理解的语言；如果我想拍卖自己的汽车，我必须使用拍卖行话。同样，作为一个编程人员，如果我想将我的 agent 加入系统，我必须入乡随俗，想与哪些 agent 交互，就要使用它们的语言和习惯来编制我的 agent。

4.3.2 AOP 语言设计的其它考虑：

1)首先，程序不可能真的和人类一样能自由行动。标准规程(例如信息搜集与发布)可以由 agent 程序处理，但重要决策应当由人来处理(agent 可以提醒主人)。因为任何应用程序都是依据一定的规则(交通规则、办公流程)编制的，程序如果不遵循一定的规则，人们就不可能接受它作为自己的代理。现实生活中的决策决非一味地循规蹈矩，往往需要违背常理、规则，甚至法律。

2)许多选择方案都是可行的，但在一个系统中不可能覆盖所有的特性(有些方面是相互抵触的)。

3)功能太多、结构太复杂的语言，在编程时很难做到透明性，因而就难以被编程人员接受。同时，利用这种语言编制出的程序，其运行效率要受到很大影响(经常发生运行超时或内存溢出)。因此，AOP 语言包含诸如演绎封闭等特性是不现实的。一个实用的语言只能涉及有限的特征，覆盖理论研究的一个很小的子集，顾及一部分任务的可处理性，而不能试图解决所有问题。

4)应注意外部观点(从理论和编程人员的角度)与内部观点(从 agent 自身的角度)之间的差别：编程人员知道系统的构成原理和有关 agent 之间的通信原语。他利用这些知识编制 agent 并确定他的 agent 如何形成信念和意图，如何决策和行动。另一方面，agent 必须具有足够的内部知识以完成自己的任务。

5)由于不同的编程人员可能在不同的时间将他们的 agent 加入系统，这要求系统的各组成部分都应具有良好的规范和文档。

4.4 AOPL 的一种实现框架

4.4.1 AOP 语言的一般组成 agent 程序应包括数据部分和过程部分。数据部分包括各种类型的心智状态：信念、意图、目标、规划、义务、能力等。心智状态的表示可以是简单的布尔值，也可以是多模态时序逻辑公式，这可以在易表示性和可处理性之间作出折衷。

数据部分涉及 agent 关于自己的知识(目标、规划等)、关于其它 agent 和环境的知识(规则、标准等)，这些知识根据内部的推理和来自外部的信息而变化。内部推理包括不同心智状态之间的相互作用，要求修改时保持知识的一致性、持久性和封闭性。

心智属性的选择要根据编程人员的意图而定。用于办公自动化的 agent 可选择义务和责任，而用于支持设计的 agent 可以选择目标和意图。

过程部分管理不同心智状态之间的推理控制、agent 之间的通信控制和执行控制，可以将其分为两种类型：一是固有规程，预定义并由解释器实现的程序模块(例如一致性检查，透明通信、时间管理、知识更新、执行控制、与其它语言程序的运行联接、PROLOG 的搜索与合一算法等)。解释器应尽量使底层协议的细节透明，使 agent 编程人员只需指明其它 agent 的名字及其主机名，就可以与其进行消息通信。不同 agent 的解释器应能依照原序传递任意的数据流。二是用户定制规程。根据 agent 规范和语言的构造手段，由编程人员实现的 agent 特定行为。

开放系统中的局域认识关系要求，agent 和它的编制者不需要知道系统中所有的 agent 和所有的语言，但有些 agent 可以在系统中充当翻译，帮助其它 agent 寻找合适的服务。

4.4.2 agent 程序的执行模式 agent 程序的执行类似于产生式系统的"识别-执行-循环"(数据识别、冲突消解、执行所选的规则)。agent 的规则可采用下述形式：

IF (belief) AND_ AND (intention) AND (time) THEN DO(action)。

agent 解释器周期一般包括三个阶段：

1)输入处理：来自其它 agent 和环境的信息(通过信息、感知或用户接口)都存储在 agent 信箱(mailbox)中。处理时，检查它们与 agent 当前心智状态的一致性，根据有关的解释器或用户定制的规程，对破坏一致性的数据进行修改(信念修正)。

2)内部处理，进一步处理心智状态。根据不同心智状态之间的推理结果产生一个新的状态，并进行有关动作的决策(从信念到动作的转化)。

3)动作输出：将有关信息送往其它 agent 并执行私有动作。

agent 可以并发工作，对紧急(特殊)消息可以激发一个并发进程而不中断当前的处理。这样，单个 agent 可能存在多个并发执行周期的重叠。但是，环境资源将限制 agent 的并发进程的多少。

4.4.3 信念修正及从信念到动作的转化 信念修正必须能反映新信息并保持局部一致性。根据信念的类型和复杂性可以采用不同的修改策略：1)严格的知识不能修改；2)有些信念只能被用户修正；3)有些信念可以被用户定制的规程修改(信念规则)；4)有些信念可以被解释器的固有规程修改。

还可以采用时间截的方法进行信念修正。每个数据都盖有一个记录它们何时被 agent 相信的时间戳，时间戳老的数据被具有新时间戳的数据取代。在一个特定时间点的推理根据该时刻成立的信念进行。

Agent 具有及时应激行为和思索推敲行为。及时应激行为由预定义或预规划的动作执行，而思索推敲行为通常采用多层规划的方式逐步执行。编制 agent 程序的中心工作是如何将信念与动作联系起来。

从信念到动作转化的一般方法是，将动作、目标、规划等的前置条件与实际的信念进行匹配，如果匹配成功，则执行相应的动作、目标或规划。这种方法多见于基于规则的系统。例如，AGENT-0 采用承诺规则、PLACA 采用心智状态改变规则、MYWORLD 采用意图采纳规则和策略规则。

但是，应注意行为定制与行为学习的区别。人类的许多行为是基于经验的，因此 agent 也应当是受经验驱动的，动作的执行也可以来自类似的实例(基于实例的推理方法)而不需要直接的外部刺激。

另一种是直接执行方法。例如在系统 CONCURRENT METATEM 中[14]，agent 通过时序逻辑公式描述，执行意味着构造公式的一个模型，及时应激行为用"下一时刻"算子○描述，思索推敲行为由"将来某一时刻"算子◇描述。

结束语 本文介绍了面向 agent 计算、编程风范以及 agent 语言设计的特点，从软件工程的角度分析了它们存在的主要问题和解决途径，并讨论了不同问题的未来研究方向。

除了用于开放系统外，面向 agent 技术还可以用于支持软件设计。在设计过程中，agent 可以解决和处理许多困难问题：谁易受设计变化的影响？谁负责解决冲突(agent 程序负责冲突检测和决策建议，最终决策由主管人员负责)？应当向谁通报决策结果等。这比起近来并发工程提供的工具来说，具有更强的功能和潜力。

参 考 文 献

[1] M. Wooldridge and N. R. Jennings, Intelligent Agents: Theory and Practice, The Knowledge Engineering Review, 10(2), 1995

[2] Y. Shoham, Agent-oriented programming, Artificial Intelligence, 1993, 60: 51-92

[3] N. R. Jennings and M. Wooldridge, Applying Agent Technology, In J. of Applied Artificial Intelligence special issue on Intelligent Agents and Multi-Agent Systems, 1995

[4] 樊晓聪等，面向 agent 软件工程(1)：规范、实现与验证，计算机科学，25(4)1998

[5] P. Sargent, Back to school for a brand new ABC, In The Guardian, 12, March, 1992

[6] Ovum Report, Intelligent agent: the new revolution in software, 1994

[7] K. P. Sycara, Argumentation: planning other agents plans, IJCAI-89, Detroit, Michigan

[8] *H. J. Muller, Negotiation principles, In Foundations of* Distributed Artificial Intelligence, O'Hare, G. and Jennings, N. R. editors, Wiley, 1995

[9] N. R. Jennings and M. Wooldridge, Software Agents, IEEE Review 42(1)1996

[10] J. Huang et al., An agent based approach to health care management, same to [3]

[11] M. R. Genesereth et al., Software Agent, CACM, 37(7)1994

[12] Y. Shoham, AGENT0: A simple agent language and its interpreter, In, AAAI-91, 1991

[13] H. D. Burkhard, Agent-Oriented Programming for Open Systems, Proc. of the 1994 Workshop on Agent Theories, Architectures, and Languages (ATAL-94), 1995

[14] M. Fisher and M. Wooldridge, Executable temporal logic for distributed A. I., In Proc. of the twelfth Intl. Workshop on Distributed Artificial Intelligence (IWDAI-93)