自适应软件的策略自动生成与演化

林华山 刘 洋 焦文品
(北京大学信息科学技术学院 北京 100871) (北京大学高可信软件技术教育部重点实验室 北京 100871)

摘 要 随着软件功能的日益强大和运行环境的日益复杂,软件要求能够及时感知环境和需求的变化,并做出相应的反应。自适应系统是一种能够通过感知环境和运用自身知识决策自身行为的软件框架。策略集是自适应系统的核心知识,但是现有自适应研究都缺少对策略的生成、维护、演化等的关注。借鉴决策树算法,提出一种使用策略树来自动生成、演化和维护策略集的方法,使得自适应软件能够更好地使用知识,排除人为制定策略对自适应软件效果的影响;同时,以RubiS网站为基础,通过仿真实验验证了方法的可行性。

关键词 自适应,策略,策略树,自动生成,演化
中图法分类号 TP301 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2017.11.002

Automatic Generation and Evolution of Policies for Self-adaptive Software

LIN Hua-shan LIU Yang JIAO Wen-pin
(School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China)
(Key Lab of High Confidence of Software Technologies (Peking University), Ministry of Education, Beijing 100871, China)

Abstract With the increment of complexities of the software features and uncertainties in the software runtime environment, software requires the ability to perceive and react to the changes from the environment in time. The self-adaptive system is a software framework with the ability to choose actions according to environment and knowledge. The set of policies plays a key role in the knowledge of self-adaptive system, but there's little research which focus on producing, management and evolution of policies. Based on the decision tree algorithm, a method of producing, management and evolution of policies was proposed, which makes the self-adaptive software perform well in using knowledge and avoiding the detrimental effect from drawing up the policies artificially. At the same time, an experiment, which is based on the RubiS website, was conducted to prove the feasibility of this method.

Keywords Self-adaptive, Policy, Policy tree, Auto-producing, Evolution

1 引言

随着软件运行环境的复杂性和不确定性以及用户需求的多样性不断上升,传统的软件已经不能很好地适应环境和需求的变化[1]。自适应软件能够根据其感知能力动态调整行为,使得软件运行倾向于满足用户需求。

自适应软件多采用MAPE-K框架实现。其中,以策略的形式表示从感知阶段的环境状态和自身状态到效应阶段执行的行为的关系映射,即策略是感知到的资源的状态到可执行行为的映射。感知效果与效应结果是由支持其运行的底层配置和硬件决定的。在软件层面,根据感知结果做出正确的决策是自适应软件最核心的能力,而策略是做出决策最重要的依据。在对自适应软件的研究中,人们往往忽略了策略的来源,而通过用户人为制定来获得软件的策略。但是,人为制定策略存在许多问题:1)无法保证策略的准确性。系统的行为在不同环境状态下可能有不同的执行结果,用户对软件系统的行为只能主观猜测,无法确定软件行为符合用户猜测。另外,部分资源的状态(如温度)是无法枚举的,如何对无限状态的资源进行划分,使得可以准确指导软件行为?人的主观划分很难保证其准确性。2)无法保证策略的完备性。运行在复杂环境下的系统,由于环境中资源数量较多,环境状态数量组合爆炸,用户无法充分认识每一个状态,无法保证制定出的策略能够覆盖所有情况。3)工作量远超用户的承受能力。在复杂系统下,可能需要更多的策略来支持软件运行,用户无法通过分析大量的信息来制定出复杂的策略体系。

为解决上述问题,本文借鉴决策树算法,提出一个基于决策树的策略框架。该框架可以通过分析软件的运行数据自动生成策略集,并且支持在软件运行中自动调整、演化、生成策略,以更好地支持软件适应环境和用户需求的变化。本文的主要贡献如下:1)提出一种自动生成策略的方法;2)运用决策树结构维护策略和驱动软件决策;3)支持软件运行过程中策略的演化。

到稿日期:2016-10-11 返修日期:2016-12-13 本文受国家重点基础研究发展计划(973)(2015CB352200),国家自然科学基金(91318301,U1201252)资助。

林华山 男,硕士,主要研究方向为自适应软件与理论,E-mail:linhuashan@pku.edu.cn;刘 洋 男,博士,主要研究方向为自适应软件与理论;
焦文品 男,教授,主要研究方向为自适应软件与理论,E-mail:jwp@sei.pku.edu.cn。

本文第 2 节介绍自适应软件和决策树的相关工作和背景；第 3 节描述一个简单的适用场景；第 4 节介绍如何使用、改进决策树，使其能够支持自适应软件的运行和策略的演化；第 5 节给出一个实例，用来展示该方法的可行性；最后总结全文并对未来工作进行展望。

2 相关工作

2.1 自适应系统

对于如何驱动自适应软件，人们做了很多研究。Cheng 团队[2-3]更关注对系统、环境等的表示；Weyns 团队[4-9]对自适应软件的相关研究做了很多整理和归纳，并提出一种灵活的框架(FORMS)来描述自适应系统[4]；Garlan 团队[19]提出的主动自适应理论使用概率模型检测的方法使得软件具有适应性，特别是在软件决策无法产生瞬间效应的复杂场景下有很好的效果。本文提出的自适应方案更侧重于探寻更佳的适应性策略而非如何选择策略。相比 Garlan 的方法，该方法在策略选择时能够快速响应，更适合于需要频繁决策的自适应软件。同时，两者关注点没有冲突，在使用决策树方案时仍能引入概率模型的决策选择方法。

文献[10]阐述了自组织系统的发展方向是分布式、动态应用，并说明了自适应软件的驱动必须考虑更加复杂的节点关系，才能有效地驱动整个分布式系统。文献[2]提出自适应系统必须能够根据环境的改变来调整自身行为，系统反馈使得系统能够更好地认识自身结构和行为，从而为系统演化提供更多可能；同时，还提出一种需求描述方式来支持不确定性和适应性的表达。文献[13]讨论了自适应软件的概念、历史、应用等现状，并提出一系列可研究的问题。

2.2 决策树和信息熵

2.2.1 决策树概述

决策树是一种常用的监督学习的分类方法。它通过对样本进行学习得到一个分类器，然后使用该分类器对新出现的对象做分类预测。每个训练样本都包括一系列属性的取值和一个分类。

决策树的算法结果是产生一个树形的分类器。不断以属性为分割标准将样本集合分割成更小的子集合，从而降低集合内的混乱程度，达到分类的目的。主流的决策树算法包括 ID3 算法和 C4.5 算法等，它们都是根据信息熵增益来选择分类属性。

2.2.2 信息熵

信息熵是一个数学上颇为抽象的概念，在这里可以被理解成衡量信息混乱程度的一个标准。信息熵越大，表示信息越混乱。

信息熵的计算公式为：

$$H(x)=-\sum p(x_i)\mathrm{lb}(p(x_i)),i=1,2,\cdots,n$$

其中，x 表示随机变量，$p(x_i)$表示变量 x_i 出现的概率。

在决策树算法中，通常使用信息熵来衡量每个分类内部的混乱程度。混乱程度高的分类需要继续做分类，直到混乱度低于可接受的阈值。每次分类必须使得分类的混乱程度降低才是有意义的。为了更快地达到低混乱度要求，往往选择信息熵增益最大的标准来进行分类。

2.2.3 信息增益

信息增益是在决策树选择分类特征时最常用的标准。一个分类方法对一个集合产生的信息增益为分类前集合的信息熵与分类后各子类信息熵加权和的差。信息增益的值越大，表示通过分类后，信息的混乱程度下降得越快。

$$G(S,A)=H(S)-\sum_{v\in V(A)}\frac{|S_v|}{|S|}H(S_v)$$

其中，$V(A)$是属性 A 的值域，S 是样本集合，S_v 是 S 中在属性 A 上值等于 v 的样本集合。

3 场景——网购平台服务器管理

为了更好地阐述本文的方法，在此提出一个简单的应用场景进行辅助说明——一个网购网站的后台服务器管理。一方面，网购网站的高峰流量大，需要投入较多的服务器才能满足用户需求。但是，网购网站的流量差也很明显，每天的高、低峰相差巨大，在低峰区间可以关闭一部分服务器来减少开支和损耗。另一方面，服务器的负载越高，需要耗费更多的电能等用于散热。因此，平衡服务器数量与每个服务器的负载是值得研究的问题。同时，由于电价的波动以及公司发展而带来的需求变动等原因，简单设计几个阈值的管理方法很不精确，不能带来很好的效果，需要根据实际情况实时调整。

管理人员根据经验对系统进行配置，形成了很多经验性的系统运行状况数据。运行数据主要为系统能够感知到的资源状态和系统采取的行为，以及行为的收益。其中，该系统能够感知到的环境资源包括时段、服务器的平均负载、平均响应时间；系统能够采取的动作包括增加服务器、减少服务器、设定等待时间等。本文主要致力于解决如何使用已有的大量运行数据来生成一个好的策略集，使得该策略集优于管理人员的经验定制，其既能够发掘出好的策略，又能规避不好的策略，使得系统有更好的适应性。

该场景的数据样例如表 1 所列。

表 1 数据样例

属性	T	N	$L/\%$	RT	A	Δ
样例数据	3	100	80	10	Down_N	1.0
	12	120	90	20	Up_N	1.2
	21	200	100	30	Up_N	1.1
	21	250	100	20	Down_L	0.5
	21	250	100	20	Down_N	−0.6

表 1 中，T 表示自适应系统的响应时间；N 表示当时服务器数量；L 表示当时服务器设定的负载上限；RT 表示平均响应时间；A 为自适应系统在当前状态下采取的行为；Δ 表示动作 A 执行前后目标值的差，代表策略的收益。

4 策略的自动生成与演化

策略集合在驱动自适应软件向满足用户需求的方向发展起着至关重要的作用。传统的策略表示方法存在如下缺陷：1)策略间的关系无法体现。策略间存在的复杂关系可以分为两类，一类是 condition(策略触发的条件)间的关系，因为 condition 所表示的状态集合存在交集或者包含等情况，使得策略间隐含复杂的触发关系；另一类是因为 action 之间存在依赖、冲突等而使得策略间存在可执行关系。当两条策略间同

时存在两种关系时，策略的选择将变得十分困难，在没有更多知识的支持下，甚至无法做出很好的触发选择。2）策略的响应时间与策略精度冲突。传统的策略表示方法是枚举出每一条策略，这需要遍历每一条策略来判断策略是否被触发，因此自适应软件每次响应的时间与策略的数量呈正相关。而策略制定得越精细和完整，能够使得软件的决策越精确，但其也就需要更多数量的策略。因此，快速响应与精确决策间存在冲突。3）策略制定比较困难，人力成本大。枚举型策略集的制定需要用户较多的参与。

本文提出使用策略树的结构来表示策略集。策略树是一个由决策树算法生成的树型数据结构。该树的每个节点都表示一个分类，根节点是初始分类，即所有的训练样本都在该分类中。每个非叶子节点都会根据算法分成若干个信息熵更小的子类，形成其子节点，父节点到子节点的边记录其分类的特征属性。一个分类达到一定条件时不再分裂出子类，而转化为叶子节点。叶子节点详细记录该分类下的样本数据的各个动作、数量、期望收益等信息，用于策略选择。当感知到系统的资源状态时，其从根节点开始，根据分类特征不断分给某一个子节点，直到将当前状态分配到某一个叶节点。然后，叶节点根据已有知识决策出系统将要采取的行为，当行为结束后，将行为结果也计入该叶节点，以丰富该节点的知识。

该方法具有以下优点：1）策略触发的复杂度只与环境中资源的数量有关。决策树的响应复杂度正相关于树的高度，从决策树的特征易知树的高度等于环境中资源的数量。2）策略间的关系得到简化，利于策略的对比和选择。每一条从根节点到叶子节点的路径都表示一条策略或多条 condition 相同的策略。因此策略间的 condition 关系只有两种，要么不相交，要么重合。因为 condition 互斥的策略在逻辑上不会被同时触发，所以它们的 action 是否存在冲突不再重要。condition 相同的策略总是同时出发，只有 action 不同，在相同条件下对它们进行评估和对比将更加准确。3）策略集可以自动生成。借助于决策树算法，在其上进行更符合策略集需求的改进，可以通过学习以往或其他系统的经验自动生成策略集，不再需要用户参与，降低了使用成本。4）支持策略集的演化。树结构可以比较方便地进行增、剪枝，特别是在增枝时，是在原有的基础上进行分裂，比创造一个完整的 condition 简单得多。

4.1　决策树的生成和改进

决策树算法是一个经典的机器学习算法，其基本思想可以用如下伪代码表示。

```
1. Decision_tree(DataSet * S, AttriList * attri_list)
2. {
3.   if(Is_leaf(S, attri_list)) {
4.     Set_as_leaf(node);
5.     return node;
6.   }
7.   Select_splitting_attri(S, attri_list, node);
8.   Remove_attri(attri_list, node->attri);
9.   foreach(attri_value in node->attri) {
10.    Construct_subset(S, node->attri, attri_value, s);
11.      if(s== null) {
12.        Vote_for_decision(S, subnode);
13.      } else {
14.        subnode= Decision_tree(s, attri_list);
15.    }
16.    Add_subnode(node, subnode);
17. }
18. Add_attri(attri_list, node->attri);
19. return node;
20.}
```

本文借鉴决策树的算法来生成初始的策略树。为了使该方法更契合 agent 的特征，需要做以下改进：

3. 判断当前状态是否为叶子节点

4. 把 node 置为叶子节点

7. 选取划分属性，并将划分信息写到 node 中

8. 删除剩余属性集中选取的划分属性

9. 根据划分属性的属性值，将全集 S 分成若干子集并分别构造决策树

10. 根据选取的属性值构造训练子集

11. 子集为空，添加叶节点，可对 S 采用多数表决法

18. 需要将删除的属性恢复，因为 attri_list 为共享区

4.1.1　分类

决策树算法本质上是一个分类算法，要求对训练数据先进行分类的标注。而 agent 系统的运行数据中没有显式的类别，因此，需要在算法执行前进行标注。

策略可以被认为是从资源状态到 action 的映射，因此资源状态是算法的变量。如果以 action 作为分类，那么决策树算法将训练出触发 action 的状态集合分类。但是，分类中不区分 action 执行的效果，如果其中有大量效果不佳的数据，那么产生的策略的执行效果也很可能不佳。理想的策略是表述在一定状态下执行某 action 可以取得正面收益，或至少是阻止执行某动作以防产生负收益。显然，用 action 作为分类是不合适的。另一方面，如果以 action 和其收益作为分类，收益情况根据评估方式和评估粒度的不同会有很大差异。如果评估粒度很小，会使得收益情况很多，直接使用 action 和收益作为分类将会导致分类数量太多，需要更多数据才能产生可靠结果；同时类与类之间的差异情况也更复杂，有一些类间差异很小，而有一些类间差异相对很大，使得算法结果的准确度较低。

本文拟采用的分类方法是：基于 action 和其收益进行分类，但是只将收益分为正收益和负收益两类。这样，分类数量即为 $2N$。同时，每个分类的特征也很明显，表示执行 action 取得正或负收益。其中，负收益的分类用来阻止软件在一定状态下执行某些动作。如果正、负收益有明显偏差，则采用平均数等统计学方法重新进行划分。

以场景中表 1 的数据为例，它们的分类情况如表 2 所列。

表 2　分类样例

属性	T	N	L/%	RT	A	Δ	C
样例数据	3	100	80	10	Down_N	1	Down_N+
	12	120	90	20	Up_N	1.2	Up_N+
	21	200	100	30	Up_N	1.1	Up_N+
	21	250	100	20	Down_L	0.5	Down_L+
	21	250	100	20	Down_N	−0.6	Down_N−

表 2 中，classification（C）表示分类，以动作名加（+/−）

来表示执行该动作后取得了正收益或负收益。

4.1.2 叶节点

传统的决策树的叶节点只保留该分类的类别信息，当结果中含有多种类别时，往往只使用最多数类别标记叶节点。但是对于策略树而言，该方法并不合适，原因如下：1)分类数量多不代表策略优。本文使用决策树的目的是得到好的策略集，在一定 condition 下(一条路径)数量多的分类不代表它就是最好的策略，如某分类中有 a1+类的数据 10 条，a2+类的数据 8 条，但不能断定在该 condition 下 a1 的效果好于 a2，因此只保留 a1 标记叶节点是不准确的。2)丢失大量策略，导致完整性差。如上例，可以认为有 condition->a1，condition->a2 两条策略，在 condition 满足时，两条策略都应该被触发，对其评估后选择一条更优的执行。如果只保留 a1 标记该叶节点，等同于每个 condition 都只有一条策略，会丢失其他竞争策略，既不完整，又丧失了自适应软件的特点。因此，本文对叶节点不采用多数代表的方法，而是记录所有的分类，分别代表一个策略；同时，保存其他统计信息用于评估，预测策略效果。

例如，$T=21$，$N=250$，$L=100\%$，$RT=20$ms 的叶节点上的内容可能如表 3 所列。

表 3 叶节点样例

属性	C	num	$profit$
策略 1	Down_L+	10	0.5
策略 2	Down_N-	4	-0.6
策略 3	Up_N+	8	1.0

其中，C，num，$profit$ 分别表示分类、数量和预测收益。叶节点的每一条策略都可以用一个三元组$\langle C,N,P\rangle$表示。而叶节点包含一个 condition 和一系列$\langle C,N,P\rangle$，因此叶节点也可以用一个二元组表示为$\langle Con,List\rangle$，其中 list 是一个$\langle C,N,P\rangle$三元组的集合。

该叶节点包含 3 条策略，分别是：

$T=21\wedge N=250\wedge L=100\%\wedge RT=20\rightarrow Down\_L$

$T=21\wedge N=250\wedge L=100\%\wedge RT=20\rightarrow\rightarrow Down\_N$

$T=21\wedge N=250\wedge L=100\%\wedge RT=20\rightarrow Up\_N$

同时，该叶节点还包含以上每条策略被执行的次数和预计执行该策略取得的收益，该预测收益是之前所有在当前状态下执行该动作的收益的平均值。在触发策略时，根据一定的规则，选择其中一条进行触发。当然，根据不同场景的需求，可以保存更多的信息用于策略选择。

4.1.3 信息熵

传统信息熵的定义认为不同的分类之间的混乱程度与信息内涵是无关的，只与其概率有关。只有分类之间彼此没有影响且彼此正交的分类满足传统信息熵的定义。例如：对随机产生的整数进行分类，奇数分类和偶数分类是正交的，以下称这种分类为正交分类。

但是在实际运用中，不同分类之间不一定是正交的，只以传统的信息熵公式进行计算并不一定能达到很好的效果。4.1.1节提出的分类方法不是正交分类，因为：1)a1+和 a1-之间存在互斥，它们分别表示 a1 动作取得正收益和负收益；2)动作之间可能本身就存在一定的依赖、互斥等关系。

为了训练出更符合自适应软件特征的策略树，对信息熵做如下改进：

$$Entropy(S)=H(S)+\sum_{a,b\in Class}T(a,b)$$

$$T(a,b)=\frac{-\cos(\vartheta)}{n-1}(-p(a)\mathrm{lb}\ p(a)-p(b)\mathrm{lb}\ p(b)$$

其中，S 是样本集合；H 是传统信息熵；a 和 b 是不同的类，$p(a)$，$p(b)$表示 a，b 类在 S 中出现的概率；$-\cos(\vartheta)$表示 a，b 类的关系。若 a，b 正交，$\vartheta=\frac{\pi}{2}$；若 a，b 相同，$\vartheta=0$；若 a，b 完全互斥，则 $\vartheta=\pi$。T 表示两个分类之间的附加信息熵。当 a，b 正交时，$T=0$；当 a，b 相同时，$T=-H$；当 a，b 互斥时，$T=H$。当 S 中的所有分类正交时，显然 $E=H$，满足传统信息熵的定义。当 S 中的所有分类相同时，$E=0$。

4.2 策略的概率触发和反馈调整

根据决策树算法自动生成的策略树并不是最优的，随着环境和用户需求的变化，原有的能满足需求的策略也可能不再满足。为了寻找更优的策略和适应环境、需求的变化，自适应系统在兼顾完成用户需求的情况下，还要演化出新的策略。

在触发的策略树叶节点上，往往不止一个 action，即不止一条策略被触发，此时需要先对不同策略做一个评估，然后选择评估最高的策略执行。在每次执行后，执行结果都需要反馈到对该策略的评估中。但是，只选择评估最高的策略执行会使得真正得到运行的策略很少，随着环境和需求的变化，基于早期数据的评估也不一定准确。为了使更多策略在实际环境、需求下获得执行机会，本文采用概率触发的机制来选择真正用于执行的策略。评估高的策略的执行概率比评分低的策略的执行概率更高，但是低分策略也有被执行的可能。

鉴于如上的考虑，本文选择的触发概率是：

$$assessment_a=\lg(num_a)*profit_a$$

$$p_a=assessment_a/(\sum_{j\in A}assessment_j)$$

其中，num，$profit$ 分别表示动作的数量和期望收益(见4.1.2节)。用 $assessment$ 值来表示对每个策略的评估，对所有触发的策略的评估值做归一化处理，得到每个策略的触发概率 p_a。

系统的每一次行为都会形成一条新的数据反馈到叶节点中，其会使得触发的动作的 num 值加 1，更重要的是，其真实的执行收益会影响 $profit$ 的值，从而改变了 $assessment$ 和 p 的值，达到反馈调整的效果。当收益较高时，p 值增大，当收益较低时，p 值减小。新的预期收益的计算方式为：

$$profit'=(num*profit+profit_{new})/(num+1)$$

其中，$profit_{new}$表示新加入的数据的收益。

以上的反馈-概率触发机制只能够在已有策略中进行调整，并不能演化出新策略。本文在概率触发中加入更多的action选择。除了原有策略可以触发外，在叶节点中没有的action或叶节点禁止的 action 也有一定概率被触发。增加这些使得在各种 condition 下做更多尝试，为演化新策略提供可能。

4.3 增、剪枝

在策略演化过程中产生新策略于策略树上有两种形式，一种是在叶节点中增加更多的可选动作，即新策略与某些原策略有相同的 condition，但 action 不同；另一种是新策略与原策略的 condition 不同，这时需要在策略树上增加新的分支来表示。当然演化过程中也会去掉一些冗余的策略，表现为剪掉一些分支或节点。

4.3.1　剪枝

虽然决策树的触发复杂度由树的深度即资源数量决定，但是过多的分支会使得树的节点数量呈爆炸性增长，从而增加系统负荷。因此，必须对策略树进行必要的剪枝。

根据剪枝作用阶段的不同，可以分为如下两类。

(1)在决策树训练时进行剪枝。在生成决策树的过程中，不需要分裂到无法分裂为止，可以通过一定的条件来停止分裂。可用方法有：1)当信息熵达到一个阈值即可停止分裂；2)当集合中分类数量少于某阈值时停止分裂；3)当集合中元素数少于某个阈值时停止分裂。

(2)在自适应系统运行一段时间后，可以进行剪枝，从而去掉一些无用策略。

1)如果一个叶子节点包含太多 action，而某些 action 的触发次数很少，评估很低，则可以去掉该 action，降低叶节点的维护成本。因为随机触发机制仍会给没有在叶节点的 action 执行的机会，所以其是否在叶节点内，对触发概率的影响不大。

2)如果某节点只有一个子节点，那么合并该节点和其子节点。

3)如果某个叶节点中的策略没有被触发，或者只有少量被触发，则可以将该叶节点合并到其左右兄弟节点。当同时有两个兄弟节点时，选择信息熵更小的节点合并。合并时，修改新节点的 condition，然后根据 condition 将旧节点的数据分配到兄弟节点的各个叶节点。

4.3.2　增枝与新策略生成

当系统运行一段时间后，如果某叶节点的信息熵明显增大，那么说明该节点反馈调整的信息与原有策略差异较大，导致这种差异的原因可能是：1)原有策略集准确度较低；2)环境、需求等发生改变，使得原有策略不再有很好的适应性。这时，需要对该节点进行调整。

调整分为两个方面：

(1)无论是哪种原因导致的信息熵增大，都说明原样本对现有情况不再有很好的指导效果，可以选择丢弃一些发生时间较久远的旧数据。

(2)仍使用决策树算法的方法，将叶节点分裂成若干个子节点，从而重新将叶节点的信息熵调整到可接受的阈值之内。但是，分类过程中可能选择已经使用过的特征属性进行分类。这时，为了保持每条路径上属性特征唯一，需要对新策略树进行转化。

5　仿真实验

根据第 3 节提出的网购平台服务器管理场景，本文以 RubiS 为基础，加入自适应相关模块后进行仿真实验。RubiS 是一个以 eBay.com 为蓝本的、用于评估应用程序设计模型和性能的网站原型。为了能够支持多个 Web 服务器，增加了一个负载控制器来分配请求到各个服务器。该系统能够采取的行为包括：增减服务器、调整服务器负债。

以下的策略树图中，将进入节点的路径作为非叶节点名称。以叶节点中优先级最高的策略为叶节点名，并不代表每个叶节点只有一个策略。其中，动作名简单地以一个字母区分。

图 1 为根据决策树方法，在本文提出的信息熵模型指导下生成的初始策略树。策略被分布在 7 个叶节点上，每个节点内有一条或者数条触发条件相同的策略。自适应系统根据感知到的环境，在决策树中找到一个叶节点触发，然后根据相应的规则从中选择要被执行的策略。

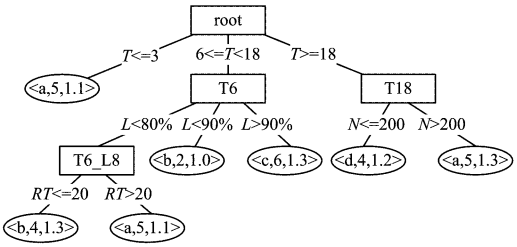

图 1　初始策略树

图 2 为运行一段时间后，对图 1 进行了一次剪枝后的策略集。其中，图 1 的叶节点〈b,2,1.0〉被触发的次数太少，根据信息熵将其与其左兄弟 T6_L8 合并成图 2 的 T6_L8。合并时，先更新了进入新节点的特征条件，然后将图 1 的〈b,2,1.0〉包含的数据分发到〈b,4,1.3〉和〈a,5,1.1〉，从而形成新的节点〈b,6,1.2〉和〈a,5,1.1〉。显然，新的叶节点与图 1 的相应叶节点不同，其可能包含不同的策略，至少策略的权重可能因为新分发进数据而发生改变。此处体现了剪枝导致的策略演化。

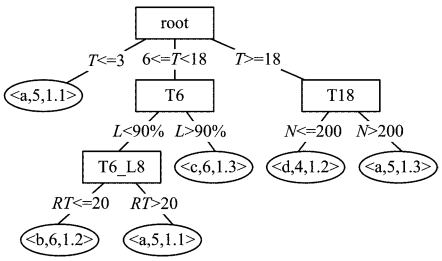

图 2　剪枝后的策略树

图 3 为运行一段时间后，对图 2 的叶节点〈a,5,1.3〉进行增枝后的策略集。其中，叶节点〈a,5,1.3〉被触发次数较多，产生一次增枝演化。根据信息熵选择信息增益最大的分类(此处为 T)将叶节点〈a,5,1.3〉分为叶节点〈a,5,1.3〉和叶节点〈c,4,1.2〉。

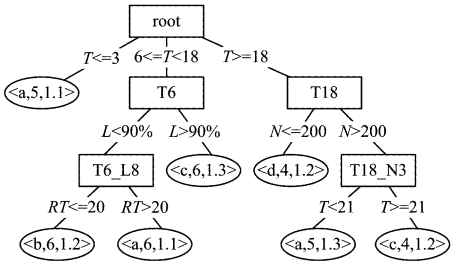

图 3　增枝后的策略树

图 3 的叶节点〈a,5,1.3〉和叶节点〈c,4,1.2〉的路径上都使用了两次 T 属性作为分类特征，需要进行调整，使得其只出现一次。图 4 为变换后的结果。

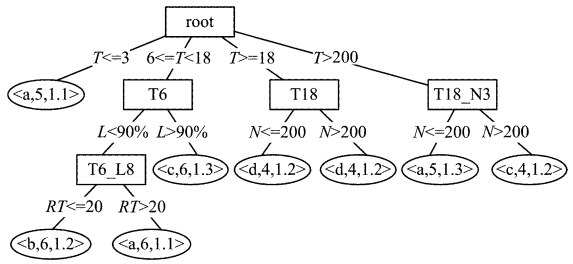

图 4 变换后的策略树

图 5 是仿真实验的效果曲线。目标满足度用 U 表示：

$$U=C\cdot G\cdot N-\Sigma E$$

其中，N 表示一段时间内及时响应的请求数(在阈值 t 时间内响应请求)；G 表示用户满意度，等于及时响应的请求与总请求数的比值；C 表示每及时响应一个请求带来的收益；E 表示一段时间内各服务器的消耗。

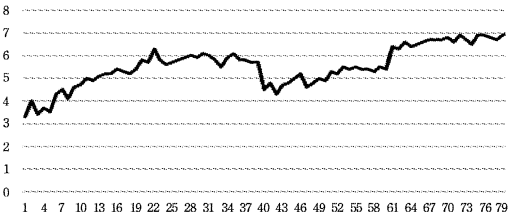

图 5 目标满足度曲线图

其中，在 40 时刻环境发生了变化，在 60 时刻需求发生了改变。由图 5 可看出，该方法能够使得自适应软件在环境、需求发生变化的情况下，仍能适应性调整自身行为，从而更好地满足用户目标。

结束语 本文关注自适应软件的策略生成、管理和演化问题，提出了一种基于策略树的自适应系统驱动方法。主要的贡献有以下几点：

1)提出策略树的策略集管理结构。该结构能够保持稳定的策略触发时间，能够保留一定的策略之间的关系。

2)提出一个策略集自动生成的算法，摆脱传统自适应软件仍由人工制定策略的弊端，提升了策略制定的准确性和完备性。

3)提出一个基于策略树的策略演化方案，使得自适应系统在运行过程中能够更好地适应环境、需求的变化，调整自身行为。

未来将在目前的策略树的基础上进一步完善策略的演化方案，使得策略演化能够更快适应更加复杂的环境和需求的变化。

参 考 文 献

[1] LADDAGA R. Guest Editor's Introduction: Creating Robust Software through Self-Adaptation[J]. IEEE Intelligent Systems, 1999, 14(3): 26-29.

[2] CHENG B H C, DE LEMOS R, GIESE H, et al. Software Engineering for self-Adaptive Systems[M]. Springer Berlin Heidelberg, 2009.

[3] CHENG B H C, DE LEMOS R, BENCOMO N, et al. Software Engineering for Self-Adaptive Systems: A Research Roadmap [M] // Software Engineering for Self-Adaptive Systems II. Springer Berlin Heidelberg, 2009: 1-32.

[4] WEYNS D, MALEK S, ANDERSSON J. FORMS: a Formal Reference Model for Self-adaptation[C]// Proceedings of the 7th International Conference on Autonomic Computing (ICAC'10). ACM, New York, NY, USA, 2010: 205-214.

[5] WEYNS D, IFTIKHAR M U, DE LA lGLESIA D G, et al. A Survey of Formal Methods in Self-Adaptive Systems[C]// Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering(C3S2E'12). ACM, New York, NY, USA, 2012: 67-79.

[6] WEYNS D, IFTIKHAR M U, MALEK S, et al. Claims and Supporting Evidence for Self-Adaptive Systems: a Literature Study [C]// Proceedings of the 7th International Symposium on Software Engineering for Adaptive and self-Managing Systems. Piscataway, NJ, USA: IEEE, 2012: 89-98.

[7] IFTIKHAR M U, WEYNS D. A Case Study on Formal Verification of Self-Adaptive Behaviors in a Decentralized System[J]. Electronic Proceedings of Theoretical Computer Science, 2012, 91(103): 45-46.

[8] IFTIKHAR M U, WEYNS D. Formal Verification of Self-Adaptive Behaviors in Decentralized Systems with Uppaal: An Initial Study[M]. 2012.

[9] WEYNS D, et al. On Patterns for Decentralized Control in Self-Adaptive Systmes[M]// Software Engineering for Self-Adaptive Systems II. Springer, 2012.

[10] D'LPPOLITO N R, BRABERMAN V, PITERMAN N, et al. Synthesis of live behavior models[C]// Proceedings of the eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, 2010: 77-86.

[11] KHAKPOUR N, KHOSRAVI R, SIRJANI M, et al. Formal Analysis of Policy-Based Self-Adaptive Systems [C] // ACM Symposium on Applied Computing. ACM, 2010: 2536-2543.

[12] PARUNAK H V D, SVEN A. Brueckner: Software Engineering for self-Organizing Systems[J]. Knowledge Engineering Review, 2015, 30(4): 419-434.

[13] LADDAGA R. Self Adaptive Software Problems and Projects [C]// International IEEE Workshop on Software Evolvability (SE'06). IEEE Computer Society, 2006: 3-10.

[14] PARUNAK H V D, et al. Software Engineering for self-Organizing Systems[J]. Knowledge Engineering Review, 2015, 30(4): 419-434.

[15] BEAL J, KNIGHT JR T F. Analyzing composability in a Sparse Encoding Model of Memorization and Association[C]// Proceedings of the Seventh IEEE International Conference on Development and Learning (ICDL 2008). 2008: 180-185.

[16] GEORGAS J C, TAYLOR R N. Policy-Based Architectural Adaptation Management: Robotics Domain Case Studies[C]// Software Engineering for Self-adaptive Systems (DBLP). 2009: 89-108.

[17] GERSHENSON C. Design and Control of Self-organizing Systems[J]. Lap Lambert Academic Publishing, 2007.

[18] LEMOS R D, GIESE H, MULLER H A, et al. Software Engineering for Self-Adaptive Systems: A second Research Roadmap [M] // Software Engineering for Self-adaptive Systems II. Springer Berlin Heidelberg. 2013: 1-32.

[19] MORENO G A, CAMARA J, GARLAN D. Proactive Self-Adaptation under Uncertainty: A Probabilistic Model Checking Approach[C]// Joint Meeting on Foundations of Software Engineering. ACM, 2015: 1-12.