软件运行时可变性动态演化机制研究

刘吉伟 毛新军

(国防科学技术大学计算机学院 长沙 410073)

摘 要 持续变化的需求对开放环境下软件系统的灵活性和可伸缩性提出了较高的要求.在许多情况下,这需要系统中能够发生变化的位置、变化的程度等可以被动态调整,即要求软件的运行时可变性是可以动态演化的.然而在多数系统(例如自适应系统)中,软件运行时可变性受限于开发者在设计阶段基于当时需求和环境要求做出的决策和假设,使得软件可变性模型本身是难以动态改变的,进而导致了系统在运行时难以适应持续变化的需求和环境.软件运行时可变性动态演化是解决问题的关键.但目前只有少数研究工作关注该方面,而且它们往往停留在模型建立和验证阶段,缺乏如何实现动态演化的研究.为解决此问题,提出了元变机制以支持对可变性要素数量、属性、关系的改变.元变机制利用体系结构动态调整技术在体系结构和构件两个层次上实现可变性演化的基本操作,即可变性要素的增加、替换和删除;又利用可变性对象实现了对这些操作的封装和信息隐藏,使人们在关注高层的可变性模型变化时无须考虑可变性要素的异构性等细节.该文给出了元变机制的基础设施并基于 .Net Framework 阐述了其生成方法,并提供了相应工具以观察或触发运行时可变性模型的变化,最后以个人云资源分享为背景,解决了资源清理案例中的运行时可变性的动态演化问题,展示了元变机制的可行性和有效性;并对元变机制的基本操作进行了性能测试,展示了其高效性.

关键词 运行时可变性动态演化;演化机制;运行时可变性;可变性对象;软件体系结构动态调整;可变性要素

中图法分类号 TP311 **DOI 号** 10.11897/SP.J.1016.2016.02216

Research on Dynamic Evolution Mechanisms of Software Runtime Variability

LIU Ji-Wei MAO Xin-Jun

(*College of Computer, National University of Defense Technology, Changsha* 410073)

Abstract Software systems situated in open environment are supposed to have high flexibility and scalability to handle ever-changing requirements. In many cases, this implies that the location where variation can occur and the degree of such variation can be changed in the systems, i.e., software runtime variability should be dynamically evolvable. However, in many systems like self-adaptive systems or other similar ones, runtime variability is limited due to design decisions and assumptions made on the basis of requirements and environments observed at design time, which typically makes runtime variability unchangeable and further results in bad adaptation of the systems to newly-changed requirements and environments at runtime. The key to the problem is dynamic evolution of runtime variability, which is concerned by few studies that only care about its logical model and validation other than implementation. The fact motivates us to propose meta-variability mechanism to support the regulation of runtime variability from an implemental perspective. The mechanism utilizes reconfiguration to realize the basic operations of evolution, i.e., in addition, substitution and removal of variability elements at level of architecture and that of component, and utilizes variability objects to support encapsulation, information-hiding

收稿日期:2015-02-12;在线出版日期:2015-11-18.本课题得到国家自然科学基金(61379051,61532004)和教育部新世纪优秀人才支持计划(NCET-10-0898)资助.**刘吉伟**,男,1987 年生,博士研究生,中国计算机学会(CCF)会员,主要研究方向为软件运行时可变性与演化、软件体系结构与设计、自适应系统.E-mail: keseter@163.com. **毛新军**,男,1970 年生,博士,教授,博士生导师,主要研究领域为软件工程、多 Agent 系统、自适应和自组织系统.

and execution of these operations. We described infrastructure of meta-variability mechanism, showed how to generate it on the basis of .Net Framework, and provided tools for observing the changes of variability model or triggering them to start the evolution. We finally showed practicability and effectiveness of the mechanism by solving runtime variability evolution in the cases of resource disposing of personal cloud. Through performance evaluation, the mechanism is proved efficiently.

Keywords　dynamic evolution of runtime variability; evolution mechanism; runtime variability; variability object; dynamic regulation of software architecture; variability element

1　引　言

软件可变性(Software Variability)是指在一定上下文中一个软件系统或软件制品(Software Artefact)被有效改变(扩展、配置、调整)的能力[1]. 在许多软件系统的开发运行阶段乃至整个生命周期中,软件可变性都是其设计开发者所要面对的基本问题[2]. 软件在运行时表现出的可变性即运行时可变性(Runtime Variability)被认为是软件应对不断演化的需求或环境的基本能力[3],近年来越来越受到自适应系统[4]、动态软件产品线[5]等软件工程领域的重视.

在大多数系统中,运行时可变性表现为软件在某个或某些可变点处对变体的绑定. 其中可变点(Variation Point)是指软件中可以发生变化(绑定变体)的位置[6],而变体(Variant)则是变化发生时,人或机器可以在相应位置做出的选择(Option). 可变点与变体统称为可变性要素(Variability Element).

例如在某基于 P2P 的网络数据传输中,为保证用户隐私安全,数据在发送之前需要使用一定算法加密. 如图 1(图例参见附录),系统设计者将抽象的加密算法(在代码层面上可能表现为委托或函数指针等)设置为可变点,将可供选择的具体算法设置为变体. 机器或人可以根据需要通过动态的绑定具体算法以在系统运行时被调用.

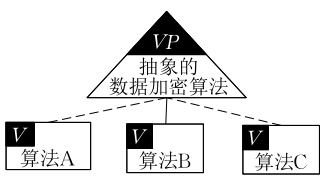


图 1　加密算法可变点及其变体

可以看出,在软件的运行实例中,变化发生的位置、变化发生的程度(决定于变体和可变性要素之间的关系)是在设计阶段预定义的. 而这种定义又在很大程度上依赖于软件的需求分析以及在此基础上做出的种种假设[7-8]. 系统虽然具有变体绑定能力来满足既定的需求,但可变性模型(可变点、变体以及它们之间的关系)本身没有变化,导致软件在应对变化的需求或环境时存在一定限制.

在上例中,假如系统安全系数提高,算法 A、B、C 都不能达到安全标准,显然仅通过传统的变体绑定(Variant Binding),无论选择哪个变体都无法使系统满足要求.

在传统意义上,软件运行时可变性模型被认为是静态的. 在许多研究工作中(如文献[9]),模型被以只读的方式访问,以支持软件适应外部的环境(见第 2 节). 人们将注意力集中在可变性绑定方面(即为什么(重新)绑定变体、绑定什么变体、怎样绑定),而忽视了对运行时可变性模型本身蕴含的限制的突破. 随着计算机技术的发展进步、用户需求的增多及变化的加快、系统驻留环境的开放、外界对系统灵活性和可伸缩性要求的提高,这种限制显得愈加突出.

软件运行时可变性动态演化是一种解决问题的方法. 它是指人或机器在运行时对软件的运行时可变性模型做出的调整①. 这种调整表现为对可变性要素数量、属性和相互关系的改变.

对于可变性要素数量的调整可以通过向当前运行时可变性模型增加新的可变点或变体,或者是删除模型内已存在的要素来实现. 比如在上例中算法 A、B、C 都不满足安全标准的情况下,就可以通过增加满足安全需求的新算法 D(变体)以使软件可在相应的可变点处绑定它,进而解决问题.

对于可变性要素属性的改变可以通过替换相应的可变性要素来完成. 如在上例中,假设人们需要在绑定算法的同时配置算法的参数,而原有可变点的

① 软件的运行时可变性可以在其他阶段被调整,比如设计阶段,它们相对运行阶段的调整(演化)而言要容易得多.

属性不能满足这个要求(如函数签名不匹配),那么其属性就需要发生改变.一般通过将其替换为具有新属性的可变点即可达到目的.在上例中,假设新增的算法 D 需要软件在可变点处通过新的函数签名来调用之,那么就可以通过替换原有可变点来达到目的.

可变性要素之间的关系可以视为相关要素的一种特殊属性.因而可以通过替换关系双方之一或全部来调整之.在上例中,假设因为资源占用原因,使用算法 B 的同时不能使用算法 C(换言之,B 排斥 C),而新需求的出现使得系统需要同时调用 B 和 C 来完成数据加密.为解决此问题,可以通过替换 B 为更为优化的算法 B′以取消原有 B 与 C 之间的排斥关系,进而使系统满足需求.

综上所述,运行时可变性动态演化是通过增加、替换或删除可变性要素这些基本操作完成的.这些操作所依赖的机制即软件运行时可变性动态演化机制(下文中简称演化机制).

对于软件可变性演化的现有研究(如文献[10-14]等)基本上停留在对于可变性及其演化的模型建立、表示[15]和验证层面[16],很少有研究工作关注如何设计出上述的演化机制以实现运行时可变性的动态演化.

本文认为该问题难点有二:首先是如何找到一种独立于可变性物理实现的一般机制.演化机制与可变性要素的实现密切相关.因为可变性要素的异构性、可追踪性和抽象性等方面(见 3.1 节的分析)的原因,在可变性演化过程中对可变性要素实施直接的操作(尤其是对可变点的操作)是较为困难的.我们需要一种对可变性要素的实现形式具有一般性的机制.

其次是什么样的基础设施能够支撑这样的演化机制.我们需要它能够在较高的抽象层面隐藏演化机制的执行细节(这些细节可能会因为可变性要素的异构性等原因而有所不同);在较低的抽象层面又能支持机制的执行.

针对这两个问题,在可变性对象和体系结构动态调整的基础上设计了相应机制(称为元变机制)及其相关基础设施.本文第 2 节分析相关研究工作,对比本文主题与解决类似问题的相关研究领域;第 3 节首先分析可变性要素的异构性、可追踪性和抽象性,而后在此基础上论述元变机制的执行过程、基础设施及其关键技术;第 4 节以个人云为背景展示元变机制对软件运行时可变性动态演化的支持并对机制作性能测试;最后一节总结全文并做出展望.

2 相关工作分析

本节通过对比自适应系统、动态软件产品线等领域的概念和思想与软件运行时可变性动态演化的不同,以进一步明确本文的问题与主要贡献.

2.1 自适应系统与运行时可变性动态演化

自适应系统(Self-adaptive Systems)可以自动地根据环境的变化来调整自身[4].换言之,它可以收集和分析有关环境的信息,基于分析的结果决定是否改变自己的结构或行为,如果是,再采取调整动作[17].这 4 个环节构成了自适应系统以及与其类似系统最基本的行为模式,也决定了自适应系统本身难以应对不断演化的需求或环境.原因如下:

系统的设计者必须指出或假设系统所面对环境变化等信息,才能确定系统所能够感知和收集的环境数据,进而以一定的数据结构将其表示在计算机内.而后才能设计相应的算法对这些信息进行分析并做出决策.设计者难以在系统部署运行之前设想所有的需求或环境的变化,当超过预期的变化出现时,已经成型的自适应系统无法自动地收集与其相关的信息,更不论其余 3 个环节.换言之,自适应技术难以解决运行时可变性动态演化能够解决的问题.

其次,在大部分情况下,自适应在执行环节对于自身的调整实际上是软件在可变点处对于变体的动态绑定,这是其运行时可变性的表现(如文献[9]).比如在图 1 的例子中,系统能够根据不同的加密级别选择绑定不同的算法.自适应并未改变可变性要素的数量、属性或关系,所以不是运行时可变性的动态演化.

再次,系统的自适应行为不需要人类干预(至少人们希望是这样).这意味着来源于自动控制理论的闭合控制环路是必不可少的.运行时可变性的动态演化需要系统开发者或维护者触发系统对可变性要素的增加、删除或替换行为,并干预其中的一部分活动.闭合环路对于系统的构建而言不是必要条件.

最后,针对自适应系统难以应对非预期环境变化的问题,有研究工作提出对系统本身进行演化的解决方法.例如文献[18],通过对感知、决策、执行等环节的封装以及体系结构的动态调整,支持对系统适应环境能力的在线调整.这种调整类似于 2.2 节提到的动态软件产品线的领域工程过程.值得注意

的是，上述的调整并不意味着闭合控制环路和自动执行，所以其已超过了自适应这一概念涵盖的范围. 实际上，运行时可变性的动态演化允许系统设计者从更为一般的意义上实现对系统的改变. 比如，它也可以应用于自适应系统，而且可以实现对系统中各个层面上可变性要素的调整，并不局限于系统的执行环节.

与自适应系统类似的是自治系统(Autonomic System)[19]. 自治系统观察的对象是其自身，目标是自我管理，具有监视、分析、计划和执行共 4 个环节组成的控制环路. 该系统实际上是一类特殊的自适应系统[20]，所以不再赘述自治技术与软件运行时可变性动态演化的不同.

2.2　动态软件产品线与运行时可变性动态演化

软件产品线(Software Product Line)包括一组核心资产(Core Assets)和从资产开发而来的一系列具有共同特征的软件密集系统(即软件产品)[21]. 它是目前可变性研究最为深入的领域.

动态软件产品线(Dynamic Software Product Line，DSPL)专注于传统软件产品线所不关注的运行时可变性，扩展了后者的概念和范畴[22]. 但是目前其关注的运行时可变性大部分都是封闭的而非开放的[22]，这意味着运行时可变性应对的是开发者在设计期设想的环境变化，导致软件产品难以应对超过预期的环境变化.

DSPL 本质上是一种建立(一系列)自适应系统的方法[23]. 如文献[22]所言，产品线自身的设计者可以通过领域工程过程和应用工程过程快速开发出具有运行时可变性的软件产品，并赋予其自主选择绑定变体的能力. 与其不同的是，运行时可变性动态演化面向的是单个系统(Single System)，而且并非针对自适应系统.

根据文献[24]的综述，软件运行时可变性并不局限于软件动态产品线，这意味着其动态演化可能发生在任何面对持续变化需求和环境的系统中. 这样的系统能够在人或软件的干预下使自身的结构或行为发生变化，而演化的目的就是调整这种变化发生的位置和变化的程度.

2.3　运行时可变性绑定机制与运行时可变性动态演化

运行时可变性绑定机制①(Runtime Variability Mechanisms，下文中简称绑定机制)描述一个运行中的软件如何在一个可变点绑定相应的变体，是软件可变性实现的核心，是上述自适应系统、自治系统、动态软件产品线等所依赖的基本技术之一，与本文问题联系相对紧密，在目前多篇文献(如文献[1,3,25-27])中有所总结和论述.

绑定机制可以按照可变性要素在软件体系结构中不同层次的实现[28]分类.

(1) 体系结构层

这一层次的绑定机制往往涉及软件体系结构的重配置或是对系统内已有构件的动态绑定. 体系结构的重配置即调整构件之间关系，重新组织系统内已有构件. 构件的动态绑定即软件在运行时根据不同的需要在具有相同接口的一组构件中选择并完成初始化和调用过程.

(2) 构件层和代码层

这两个层次的绑定机制涉及对构件内部结构和行为的调整. 同软件体系结构类似，调整构件内部结构也包括对子构件(或者更小粒度的软件实体)的重配置和动态绑定. 对构件行为的调整往往通过重新组织构件的工作流或是重新设置构件内部参数来实现.

通常绑定机制并不改变可变性要素的数量、属性和关系. 但有一种例外：重新组织系统内部的软件实体可以用于动态生成新的变体(即 3.1 节的组织型变体). 在图 1 的例子中，以不同的顺序和参数组织多种加密算法可以在运行时生成新的加密过程. 这些过程具有一致的接口，可以作为变体被相应的可变点绑定和调用. 这是绑定机制用于软件可变性动态演化的一个特例. 在本文中，该绑定机制被视作增加组织型变体的一种方法.

3　支持运行时可变性动态演化的元变机制

本节根据可变性要素的异构性、可追踪性和抽象性指出了可变性要素的不同实现形式，以及元变机制所采用的技术，而后描述了元变机制在软件体系结构和构件两个层次上对可变性要素的操作过程，最后论述支持元变机制实现的关键技术.

3.1　基于可变性要素的异构性、可追踪性和抽象性的技术考虑

3.1.1　可变性要素的异构性、可追踪性和抽象性

(1) 异构性

可变性要素的异构性表现在：在逻辑层面上同

① 这里的“绑定机制”是为了在术语方面与“演化机制”相区别，按照英文 Runtime Variability Mechanisms，“绑定”一词可以略去.

一个系统中的两个可变性要素,在代码层面上可能具有不同的实现.比如,因为忽略实现细节,在逻辑层面上的两个变体可能仅仅是名称等属性不同;而在实现层面两者的实现形式可能完全不同.具体可参见下文对可变性要素实现形式的分类.

由于编程语言、运行环境等因素,可变性要素的实现形式是多种多样的.这导致与其实现形式相关的增加、删除和替换操作(如果这些操作存在且可行)也是具有不同的实现细节.如果直接采用这些操作来完成演化,软件内部需要包含所有不同类型操作的代码,开发和维护一个这样的软件很可能会消耗较大的成本.

(2)可追踪性

在软件可变性的上下文中,可追踪性考察逻辑层面上的可变性要素在软件中的物理实现以及在逻辑(符号)层面上表达这种实现的难易程度.因为模块化程度等原因,一个变体可能被非显式的表达为一个代码片断(如一个函数的一部分),也有可能被实现为构件的一部分(如一个构件内部的软件实体),或者是一个构件,甚或是由多个构件组成的(如一个子系统)①(如图 2).其对应的可变点也可能存在同样的情况.

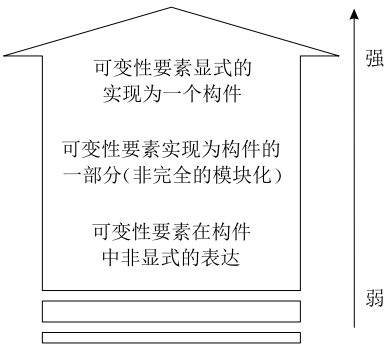


图 2 可变性要素在体系结构层面的可追踪性

一般而言,可变性要素对应的软件制品(指软件生命周期中人工制造出的产物,如代码片段)的模块化程度越高,其可追踪性越高,对增加、删除或替换操作的支持也可能越好.但出于其他方面(如开发成本)的考虑,并不是每个可变性要素都实现为模块化的软件制品.这在一定程度上导致难以直接对可追踪性差的可变性要素进行演化操作.

(3)抽象性

从可追踪性角度而言,变体可以被实现为具有独立意义的软件实体,但不一定是可以被动态增加或删除的构件.

可变点的抽象性比变体高.而可变点在体系结构层面上往往不具有直接的实现形式而只有抽象的表现(见图 3).可变点所在的抽象层次使得很难在物理实现层面上将可变点从软件实体中独立出来,形成关注点分离.这个性质导致直接实现可变点的增加、删除或替换较为困难,而且往往需要人的参与[31].这是软件运行时可变性动态演化需要人干预的原因之一.

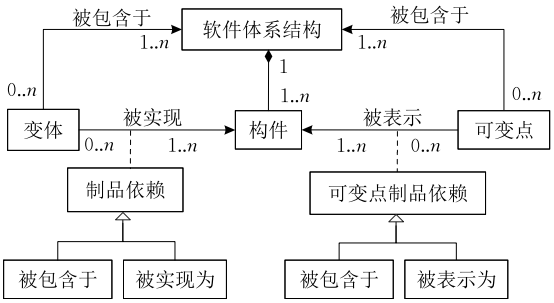


图 3 可变性要素与软件体系结构的关系

在大多数情况下,由于可变性要素的可追踪性和抽象性,并不是所有的可变性要素都可以通过与其实现形式直接相关的操作来完成可变性演化.即使这些直接操作都存在且可行,由于可变性要素的异构性,系统需要包含其所有种类操作的代码描述,会带来较大的开发和维护成本.综上所述,在可变性演化过程中直接操作可变性要素是较为困难的.

3.1.2 可变性要素的实现形式

根据上一节对可变性要素的异构性、可追踪性和抽象性的讨论,可变性要素在软件体系结构中的 3 个层次(体系结构层、构件层和代码层[28])内存在不同的实现形式.

可变性要素的实现意味着可变点可在体系结构内被表示,或变体可在体系结构内被实现.体系结构中最根本的要素即构件[32],即具有预定义接口的一段代码,可以对外提供封装在其内部的功能[33].然而构件与可变性要素之间不存在一定的关系,可变性要素具有异构性.

可变点和变体都可在软件体系结构层、构件层和代码层内被表示或被实现,如图 3.属于制品依赖(Artefact Dependency)[2]的“被包含于”关系是指(构件层或代码层的)变体可在构件内部被实现;可变

① 一个可变性要素究竟实现为什么软件实体决定于设计者按照用户需求和系统需求作出的决策.这可以参考文献[29]关于特征(Feature)与体系结构(或代码)之间的关系.特征表示系统应当具有的某方面的特点或能力[30],常用来描述需求.

点制品依赖(Variation Point Artefact Dependency)[2]的“被包含于”关系是指(构件层或代码层的)可变点可在构件内部被表示.

值得注意的是,根据对可追踪性的讨论,直接表示可变点或直接实现变体的软件制品并不一定是(完全)模块化的,然而我们总能找到一个粒度较大的模块化的软件制品(例如一个构件),其包含该软件制品,能够运行该软件制品的实例.这是图 3 中各类“被包含于”关系的确切含义.在下文中,为论述简洁,我们说“某可变性要素被实现为某软件制品”,即指该可变点或变体被直接表示为或被实现为该软件制品,或在后者内被表示或被实现.

演化机制与可变性要素的实现紧密相关.不同的实现形式意味着不同的演化机制.按照可变性要素对应的绑定机制,可变性要素的实现形式可分为选择型和组织型,每种形式可以根据其实现进一步细化分类,如图 4 所示.

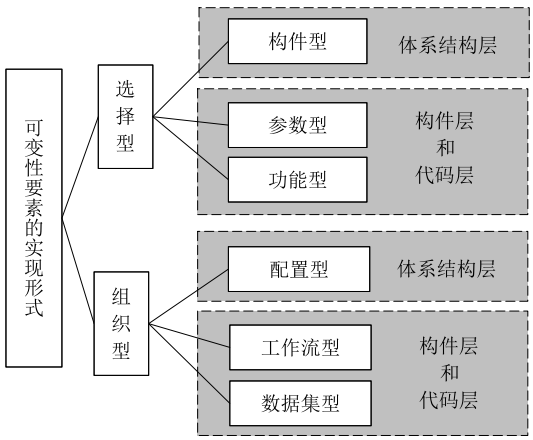


图 4　可变性要素的实现分类

选择型可变性要素通常具有特定的语义,不能通过组合系统内其他的可变性要素或软件制品来生成.例如在体系结构层内,此类可变性要素可能属于构件型.其可变点表现为构件接口,变体实现为满足该接口的构件.软件可以在诸多变体中选择一个进行绑定.

组织型可变性要素对应的变体是通过组织系统内已有的软件实体形成的.换言之,一个变体实际上是软件实体的一种组织形式(在实现层面可能是描述性的数据,不包含程序).例如在服务组合的上下文中,不同的变体即原子服务的不同组合,相应的可变点表现为可对它们进行调用的接口.

两种类型的可变性要素的细化分类参见表 1,其中可变性要素的具体实现与软件运行平台和编程语言等方面有关,表 1 中仅为举例.

表 1　可变性要素的细化分类及举例(以 C# 语言为背景)

实现形式	释义	可变点表现形式	变体实现形式
构件型	系统内具有接口和特定功能的一部分,该部分可被动态替换为其他具有相同接口的软件实体	构件接口	构件
参数型	用于调整构件或次级软件实体行为的变量	参数(如函数参数)	值
功能型	数据的处理过程,该过程可被动态替换为不同的具有相同输入输出的过程	功能接入点(如委托)	实现功能的代码段(如函数体)
配置型	系统内构件的组织形式,构件可被动态的组织成不同的形式	构件接口	构件的组合体(或描述组合信息的数据)
工作流型	数据处理过程的组织形式,过程可被动态的组织成不同的形式	功能接入点	数理处理过程的组合体(或描述组合信息的数据)
数据集型	系统内数据的组织形式,数据可被动态的组织成不同的形式	抽象的数据类型(如抽象类)	数据类型(如实现类)

3.1.3　技术考虑

元变机制对技术的采用主要考虑两个方面:一是运行时可变性模型的表示;二是增加、替换、删除可变性要素操作所依赖的方法.

软件中对运行时可变性模型进行表示是为了方便在演化过程中对可变性要素信息的表示和管理.在本文中,模型表示的意义在于在较高的抽象层次隐藏对可变性要素操作的细节(如可变性要素的异构性带来的操作方法的不同),并在较低层次支持演化机制的执行(即增加、替换、删除某类型的可变性要素).这样可以便于系统开发者或管理者考虑相对于演化机制而言更高层次的问题.

本文利用运行时可变性对象来进行模型表示,即将演化的对象,也就是可变性要素的数量、属性、关系表示为对象实例中的数据参与可变性演化的运算.具体地说,我们使用一组基类描述可变性模型逻辑上信息(如可变性要素间的关系),又在其子类对父类方法和属性的精化中又描述了与可变性要素的具体实现相关的信息(如如何实现变体绑定,如何增加、替换、删除此类型的变体).3.3.1 节对可变性对象的基类及其精化过程进行了详细论述.可变性对象与运行时可变性模型及可变性要素的实现关系如图 5 所示.

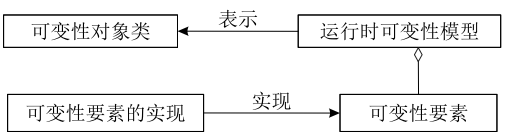


图 5　可变性对象与其实现、可变性模型、可变性对象类之间的关系

根据 3.1.1 节的结论,需要找到一种独立于可变性要素实现形式的方法直接对可变点和变体(例如数据集型的可变点和变体)进行增加、替换或删除.本文采用体系结构的动态调整这种间接的方式.它不考虑可变性要素的具体实现,仅在选择型和组织型两种抽象类型上作区分.3.2 节对此有详细论述.

3.2 元变机制概述

3.2.1 元变机制基础设施

元变机制基于可变性对象在体系结构层和构件层支持运行时可变性的演化.如图 6,在构件实例中运行有可变性要素对象.每一个可变性要素对象(某类运行时可变性对象的实例)都与构件中某个可变性要素的实现(包含于软件制品)关联.它收集关于该可变性要素诸如名称、实现形式等信息,并具有增加、删除、替换该类型要素的方法.这些方法可以调整可变性要素之间的关系,但无权直接调整软件的体系结构配置.配制调整需要构件级可变性对象群向体系结构级可变性对象群发出请求,要求动态调整软件体系结构以间接作用于实现要素的软件制品,进而实现运行时可变性的演化.

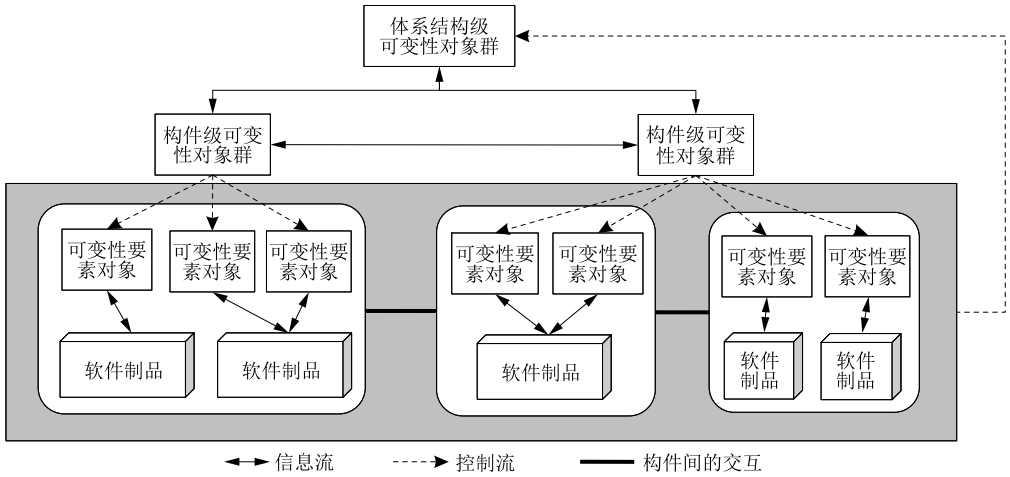


图 6 元变机制的基础设施

在实际的软件运行中,每一个具有运行时可变性的构件实例都关联一个构件级可变性对象群,籍以管理该构件内(构件层或代码层的)可能包含的大量可变性要素对象.同时可变性对象群也有汇总可变性要素信息的功能.在可变性演化的过程中,它负责调用可变性要素对象的方法以实现对可变性要素的操作.

软件运行实例在体系结构层次也包含至少一个可变性对象群.后者负责管理和调整软件系统的体系结构配置,也就是构件之间的关系形成的拓扑结构.每一种配置都被视作一个组织型变体,而可变性对象群则具有一个相应的可变点.它会根据构件层可变性要素对象的请求或人类的指令选择其中之一进行绑定(即加载配置).如果请求的配置不存在,它会利用 2.3 节介绍的方法自行生成一个新的变体并绑定.

体系结构级和构件级的可变性对象群之间存在交互.这种交互与构件间的交互是垂直的,换言之,它们利用不同的通信渠道,不相互影响.可变性对象群之间的交互大部分是通知对方体系结构配置发生变化,也可能是构件级可变性对象群向体系结构级可变性对象群发送调整体系结构的请求.

3.2.2 元变机制执行过程

3.2.2.1 软件可变性模型及元变机制执行过程符号化表述

上述文段中提到的软件体系结构动态调整包括增加新构件、替换或删除已在系统中的构件.为简化问题,本文假设构件之间只存在调用/被调用的关系.下文以符号说明简述理论上的软件可变性模型及其增加、删除、替换可变性要素的一般过程.

定义软件系统的可变性模型 M 为元组$\langle E,R\rangle$,其中 $E=\{e_i \mid i=1,2,\cdots,n\}$为可变性要素的集合,$R=E\times E$ 为要素之间关系的集合.$E=P\cup V$,其中 P 为可变点的集合,V 为变体的集合,$P\cap V=\varnothing$.

$P=\{p_0,p_1,\cdots,p_m\}$,$p_i(i=1,2,\cdots,m)$定义为元组$\langle s_{\text{name}},s_{\text{id}},t_{\text{rtype}},V_{\text{opt}},V_{\text{man}},V_{\text{bind}},E_{\text{req}},E_{\text{ex}}\rangle$,其中 s_{name},s_{id}和 t_{rtype}分别是指可变点的名称、标示符、实现类型;V_{opt},V_{man},V_{bind},E_{req}和 E_{ex}分别是可变点的可选关联变体、强制关联变体、已绑定变体、依赖的可变性要素、排斥的可变性要素的集合.

$V=\{v_0,v_1,\cdots,v_n\}$,$v_j(j=1,2,\cdots,n)$定义为元组$\langle s_{\text{name}},s_{\text{id}},t_{\text{rtype}},P_{\text{ass}},P_{\text{bind}},E_{\text{req}},E_{\text{ex}}\rangle$,其中 s_{name},s_{id},

t_{rtype}分别是指变体的名称、标示符、实现类型；P_{ass} P_{bind}，E_{req}和 E_{ex}分别是变体关联的可变点、绑定自身的可变点、依赖的可变性要素、排斥的可变性要素的集合.

显然，$\forall r\in R, r=\langle e_s, e_o\rangle$，如果 $e_s\in P$，那么$e_o\in e_s.\{V_{opt}\cup V_{man}\cup V_{bind}\cup E_{req}\cup E_{ex}\}$；如果 $e_s\in V$，那么 $e_o\in e_s.\{P_{ass}\cup P_{bind}\cup E_{req}\cup E_{ex}\}$.

令 $C=\{c_i\mid i=1,2,\cdots,m\}$为组成系统的构件的集合，有 $\forall e\in E, \exists c\in C, e\prec c$. 其中 $e\prec c$ 指构件 c 实现了可变性要素 e.

定义向 M 增加可变性要素 $e_a(e_a\notin E)$的一般过程 $f_a:\{M\}\times\{e_a\}\rightarrow\{M_a\}$，其中 $M_a=\langle E_a, R_a\rangle$，$E_a=E\cup\{e_a\}$，$R_a=E_a\times E_a$. 假设 $e_a\prec c_a$，$c_a\notin C$，取 C_a，$\forall e\in E_a, e\prec c, c\in C_a$，有 $C_a=C\cup\{c_a\}$（增加构件 c_a）或 $C_a=(C-\{c_s\mid c_s\in C\})\cup\{c_a\}$（替换构件 c_s为 c_a，此时有 $\forall e\in\{e'\mid e'\prec c_s\}, e\prec c_a$）.

定义从 M 中去除可变性要素 $e_r(e_r\in E)$的一般过程 $f_r:\{M\}\times\{e_r\}\rightarrow\{M_r\}$，$M_r=\langle E_r, R_r\rangle$，$E_r=E-\{e_r\}$，$R_r=E_r\times E_r$. 假设 $e_r\prec c_r$，$c_r\in C$，取 C_r，$\forall e\in E_r, e\prec c, c\in C_a$，有 $C_r=C-\{c_r\}$（删除构件 c_a）或 $C_r=(C-\{c_r\})\cup\{c_s\}$（替换构件 c_r为 c_s，此时有 $\forall e\in\{e'\mid e'\prec c_s\}, e\prec c_r$但是 $e_r\prec c_s$不成立）.

替换 M 中可变性要素 e_s为 $e_s'(e_s\in E, e_s'\notin E)$的过程 $f_s:\{M\}\times\{e_s\}\times\{e_s'\}\rightarrow\{M_s\}$可被视为前两种过程的组合，即 $f_s(M,e_s,e_s')=f_r(f_a(M,e_s'),e_s)$，或 $f_s(M,e_s,e_s')=f_a(f_r(M,e_s),e_s')$.

3.2.2.2　元变机制执行过程物理实现概述

物理实现层面上的元变机制执行过程（即演化过程）会因可变性要素的实现类型而稍异于一般过程. 下文按可变性要素的实现形式、操作类型（包括增加、替换、删除）简述动态调整软件体系结构如何达到演化的目的.

（1）增加选择型变体

在运行的软件实例内增加构件型或功能型变体通常包含两个步骤：向软件体系结构增加实现该变体的软件制品，建立该变体与其他要素之间的关系以使它有效（Active）.

对于构件型变体而言，实现它的软件制品就是构件本身，因此直接向软件系统增加该构件实例（连同其中的可变性要素对象，具体实现见 3.3.2 节）即可完成第一个步骤. 对于功能型变体而言，软件制品的增加可以通过增加包含其的构件来完成，也可以通过替换原系统中相关构件为实现该变体的构件来达到目的，如图 7 所示.

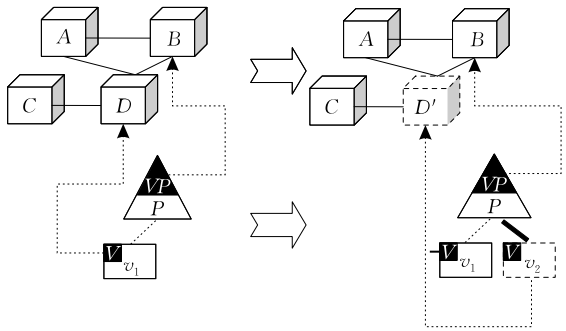


图 7　通过将构件 D 替换为 D'以增加变体 v_2

第 2 个步骤中变体有效是指直接实现变体的相关代码实例在软件的运行期内有机会得到执行. 建立它与其他可变性要素之间的关系，尤其是它与相关可变点之间的可变性依赖，可以使可变点知悉它的存在并能够选择和绑定它.

（2）替换或删除选择型变体

替换构件型或功能型的变体的步骤有三：向软件体系结构增加实现新变体的软件制品，取消原有变体与其他要素之间的关系以使其无效（Inactive），建立新变体与其他要素之间的关系以使其有效. 由于原有变体可能正在被调用，不建议去除实现原有变体的软件制品. 删除该类型的变体也遵循类似的原理，即取消原有变体与其他要素之间的关系以使其他要素（尤其是其关联的可变点）无法感知它，这样该变体就相当于被删除了.

（3）增加、替换、删除组织型变体

配置型和工作流型的变体通常是纯粹的数据. 针对此类变体的操作一般需要增加、替换或删除可变性要素对象连同相关数据，并改变可变性要素之间关系（这个方面与选择型变体操作是类似的，不再赘述）. 为了使新变体有效，其增加或替换可能涉及对软件体系结构的调整.

数据集型变体本质上是编程语言上下文中的类型（Type），与之相关的运行时可变性在动态类型语言[34]（如 Python）领域有深入的研究. 在一般意义上，增加、替换或删除此类变体可能需要通过替换相关的构件来实现. 在这个过程中，可变性要素之间关系也需要作类似于选择型变体操作的调整.

（4）增加、替换、删除选择型或组织型的可变点

为实现可变点的增加，可将在其中没有该可变点表示的相关构件替换为新的具有该可变点表示的构件. 如果该可变点可绑定的变体数目下限不是 0（即不是可选（Optional）可变点），还需要增加一个默认的与之关联和绑定的变体，以使软件实例能够

在该可变点运行实现该变体的代码，进而使该可变点是有效的.

替换构件也是实现替换可变点的一种可行的方式.与增加可变点不同的是，这个替换过程可能涉及新旧可变点之间的状态迁移(State Transformation)，即旧可变点的状态(如其当前绑定的是哪一个变体)经过转换后成为等价的新可变点的状态.

删除可变点可以采用较为轻量级的方式.事实上，如果按照上述方法将待删除的可变点当前关联但未绑定的变体删除，可变点就只能选择当前绑定的变体.如果再取消该可变点与其他可变性要素之间关系，该可变点就相当于被删除了.

3.3 关键技术

本文涉及基于.Net Framework 对上述元变机制的实现.本节主要阐述其关键技术，包括运行时可变性对象、软件体系结构动态调整以及用于辅助建立元变机制基础设施的代码生成技术.

3.3.1 运行时可变性对象

运行时可变性对象包括 3.2 节提到的可变性要素对象和可变性对象群.可变性要素对象对应的基类是根据 3.2.2.1 节的可变性模型设计的，其包括可变点对象类(RTVariationPoint)、变体对象类(RTVariant)和可变性对象群(RTVarObject-Group)，如图 8 所示.

可变点对象类和变体对象类统称为可变性对象(RTVarObject)类，用于表示和操纵软件中可变性要素及其它们之间的关系.可以观察到它具有可变点和变体的共同属性(见 3.2.2.1 节)，如名称(Name)、实现类型(VarType)、依赖的可变性要素的集合(RequiredElementList，列表类型，元素类型为对 RTVarObject 的引用)、排斥的可变性要素的集合(ExcludedElementList).

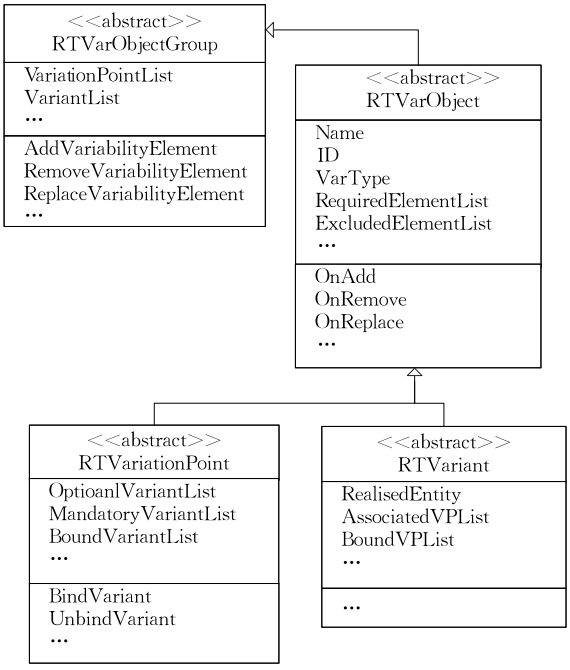


图 8 可变性对象基类

如图 9 所示，变体绑定(BindVairant)等活动都由此类(或子类)的方法描述.可变性要素对象类还具有一组抽象方法 OnAdd、OnRemove 和 OnReplace，分别用以描述如何处理可变性要素的增加(Add)、删除(Remove)和替换(Replace)对系统可能造成的影响(这种影响通常与相关可变性要素的实现形式相关).这些方法会在系统利用软件体系结构动态调整完成演化主要过程之后被调用，完成某些后续工作(例如改变自身的关系属性(如 RequiredElementList)以正确反映当前元素之间的关系等)以维护系统的一致性.

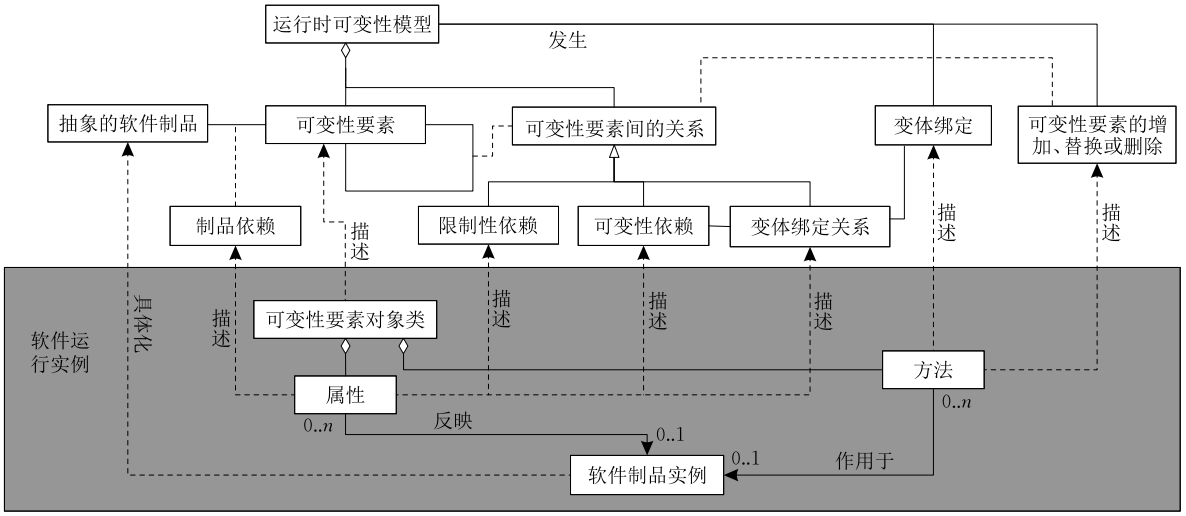


图 9 运行时可变性模型在软件运行实例中的描述

RTVariationPoint和RTVariant分别针对3.2.2.1节模型中可变点和变体的特点具有各自的属性和方法，如可变点的OptionalVariantList，即可选变体的集合(列表类型，元素类型为对RTVariant的引用)；MandatoryVariantList，即强制变体的集合；BoundVariantList，即已绑定变体的集合；再如变体的RealisedEntity是对实现变体的软件制品的引用，该属性可以用于对变体的追踪；AssociatedVPList记录关联该变体的可变点的集合；BoundVPList记录绑定该变体的可变点的集合.

为了利用可变性对象类实现对具体系统中可变性要素的表示，开发人员需要在代码中添加对上述基类RTVariationPoint和RTVariant的继承以具体描述不同种类的可变点和变体.在此过程中，开发人员需要指出可变性要素的名称、标示符、实现形式等属性值，也需要实现基类的抽象方法，尤其是OnAdd，OnRemove和OnReplace方法.多态会保证系统开发者或维护者在触发运行时可变性动态演化的过程后无须关心元变机制如何处理与异构的可变性要素相关的问题.需要注意的是，相互关联的可变点与变体的实现形式必须是匹配的，例如假设某可变点实现为一个参数，那么其关联的变体必须是具有参数类型的值.

一般的，实现构件层和代码层可变性要素的软件制品、其对应的可变性要素对象子类被封装在同一个程序集(Assembly)[35]中.在运行时，可变性要素对象的属性(如RealisedEntity)中存储软件制品的句柄(Object类型对象).对象通过它可以直接访问实现要素的软件制品的信息.除了变体绑定外，可变性要素对象不直接参与软件制品的功能执行、交互和其他软件实体的运行.理想情况下，如果在程序集中除去有关可变性要素对象的代码，其余的功能性代码应当能够正常运行.

一般的，可变性对象群用于集中管理和控制一组可变性要素对象，其中包含后者信息的列表(VariationPointList和VariantList)等组成部分，如图8所示.它扮演运行时可变性动态演化过程中执行者的角色.

其中交互代理(Proxy)负责可变性对象群之间的交互(如通知对方软件系统体系结构发生变化).具体地，它使用Web服务与处于其他主机上的可变性对象群进行远程通信，使用命名管道(Named Pipe)[35]与本地的可变性对象通信.

可变性对象管理器有3种功能：(1)汇总其管理的可变性要素对象的信息(主要是地址信息)，并将其存入可变点信息列表或变体信息列表；(2)管理可变性要素对象的生命周期，可以按需要启动或卸载对象实例；(3)根据人类的指令执行增加、替换和删除可变性对象的操作(其过程中利用多态调用了可变性要素对象子类重载的OnAdd，OnRemove和OnReplace的方法).

状态迁移器在替换操作中被用来实现旧可变性对象状态向新可变性对象状态的转化(具体过程见3.3.2节).它会提取旧可变性要素对象的属性信息，按照由程序员提供的状态迁移脚本，将数据转化为新可变性要素对象的属性值.图10中展示了状态迁移器向新增的可变性要素对象进行数据输入.

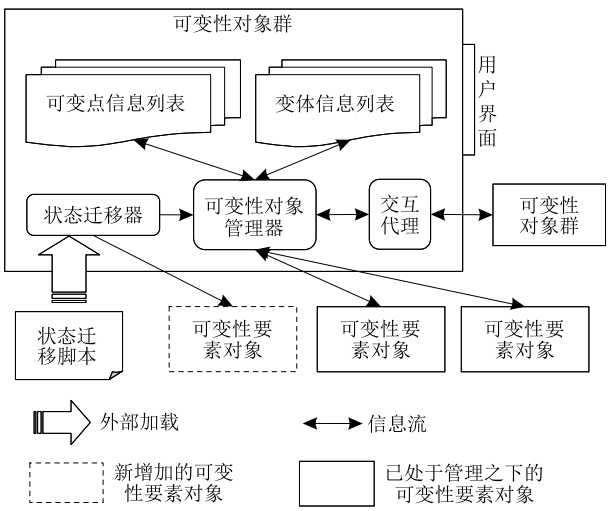


图10　可变性对象群的体系结构

构件级可变性对象群类可以位于独立的程序集中，也可以与其管理的可变性要素位于同一个程序集中.在执行可变性演化时，它可以通过交互代理向体系结构级可变性对象群发出调整软件体系结构的请求.

体系结构级可变性对象群以及其管理的可变性要素对象被封装在一个独立的程序集中.体系结构级可变性对象群通常不包含状态迁移器，因为很少出现替换配置变体的情况，而且配置本身一般不具有状态.

3.3.2　软件体系结构动态调整

理论上，元变机制并不依赖于某种特定的构件模型和体系结构动态调整技术，其只要求后者能够实现构件的增加、替换和删除，并允许在此过程中对可变性模型按演化目标进行相应的改变①.除此之

① 从一般性角度考虑，除了运行时可变性动态演化之外，设计者采用何种技术架构软件并使其体系结构可被动态调整还取决于系统需求、运行环境、系统分布、开发人员喜好等方面.

外,伴随着体系结构调整的往往还有状态迁移[36](包括可变性状态的迁移),以维持系统的运行.在实现上,由于本文采用.Net Framework 构建元变机制的基础设施(见 3.3.3 节),所以除上述要求外,还需要构件的运行实例能够与.Net Framework 互操作.下文以变体的增加为例阐述元变机制中构件增加和替换的一般过程.

过程 1. 元变机制中以增加变体 V 为目的增加构件 C 的过程.

1. 加载 C 至软件进程空间,初始化构件实例;

2. 查找并获取目标可变性对象群 G,该可变性对象群管理需要增加变体的目标可变点 P;

3. 生成并初始化构件实例中包含的所有可变性要素(包括待增加的变体 V)对应的可变性要素对象,并将其向 G 注册,使其置于后者管理之下;

4. 通过修改 P 和 V 的属性关联 P 和 V;

5. 生成包含新构件 C 的体系结构配置(变体),并向体系结构级可变性对象群注册,这个步骤实际上实现了在体系结构层面增加和绑定组织型变体的过程.

过程 2. 元变机制中以增加变体 V 为目的替换构件 C 为 C'的过程.

1. 加载构件 C'至软件进程空间,初始化构件实例;

2. 查找并获取目标可变性对象群 G',该可变性对象群管理需要增加变体的目标可变点 P;

3. 初始化 C'实例中包含的所有可变性要素(包括待增加的变体 V)对应的可变性要素对象,并将其向 G'注册,使其置于后者管理之下;

4. 通过修改的 P 和 V 属性关联 P 和 V;

5. (可选过程)提取 C 的运行状态(包括每个可变点的状态),并转化为 C'的运行状态;其中 C 的资源如内存、硬件句柄等都由 C'接管(这个步骤需要人参与以保证语义一致性,参见作者前期工作[37]);

6. 查找并获取可变性对象群 G,该可变性对象群管理 C 中所有可变性要素对应的可变性对象;反注册后者并终止其运行;

7. 释放 C 实例,终止其运行;

8. 生成包含新构件 C'的体系结构配置(变体),向体系结构级可变性对象群注册并绑定之.

在过程 2 中,除了待增加的变体及其实现,C'的其他部分应当与 C 一致,以最大程度的减小可能的维护成本,尤其是 C 中实现的可变性元素也需要在 C'中实现,C 中可变性要素本身的状态也需要通过状态迁移与 C'中的状态保持一致.可变性要素状态迁移的基本步骤为:

过程 3. 可变性要素 e 从构件 C 到 C'的状态迁移.

1. 利用.Net Framework 提供的 API 将构件 C 上下文中的 e 对应的可变性要素对象序列化为二进制码;

2. 在 C'的上下文中反序列化二进制码,生成可变性要素对象,并将该对象与 e 在 C'中的实现关联;

3. 调用 C 的上下文中的可变性要素对象的方法 OnReplace,该方法会提取 e 在 C 中的实现的状态,并执行程序员修改的 Python 脚本以转化其为 e 在 C'中的实现的状态.

步 2 中,程序员在编写脚本时需要决定哪些状态需要转化,需要怎样转化.相关的可变性对象群会自动生成状态迁移 Python 脚本,并将新旧可变性要素的状态变量按名称类似程度和数据类型对应在一起表述,以减少程序员的工作量.程序员需要指明脚本描述的哪些转化是不必要的,或者哪些转化需要类型转换等等细节,程序员需要负责新旧状态的语义一致性,通过对脚本的修改指出必要的转化步骤.详见本文之前的工作[37].

以可变点增加或替换为目的而进行的构件替换与上述过程 2 类似,不同的是步 4 需要改为保证新增可变点有效(即具有一定数量的关联的变体)的步骤.

可变性要素的删除通常不涉及体系结构的动态调整;同时,元变机制中需要对构件进行删除操作的情况比较罕见,所以不再赘述.

3.3.3 元变机制基础设施生成

元变机制的基础设施主要包括处于体系结构层和构建层的可变性对象群及其管理的可变性要素对象、与这些可变性对象群进行交互的人机界面等.基于.Net Framework 技术框架,我们提供了一系列可复用的类库和工具支持可变性要素对象和可变性对象群的构建过程.

如图 11 所示,系统设计者在编译阶段除了需要提供构件的源代码之外,还需要利用可变性要素对象信息编辑器编辑一个描述构件中所实现可变性要素对象的基本属性信息(如名称、实现形式等).

可变性要素生成器会自动将用户编辑的信息中的每一项都转化为继承可变性要素对象类的子类的属性值,而类的方法(描述变体绑定、增加/删除/替换可变性要素)是在原型方法的基础上生成的.原型方法的代码需要根据要素的实现形式从元变机制代码库中查询并提取出来,如果查询无果,生成器会要求设计者在类的方法或代码库中添加相应代码.

构件级可变性对象群的源代码是在代码库中的原型类的基础上根据用户编辑的定制信息(主要包括可变性对象群最初包含的可变性要素对象的 ID,关联的体系结构级可变性对象群信息等)生成的.可变性对象群生成器会在其初始化代码中关联指定的可变性要素实例,以形成对要素的管理.

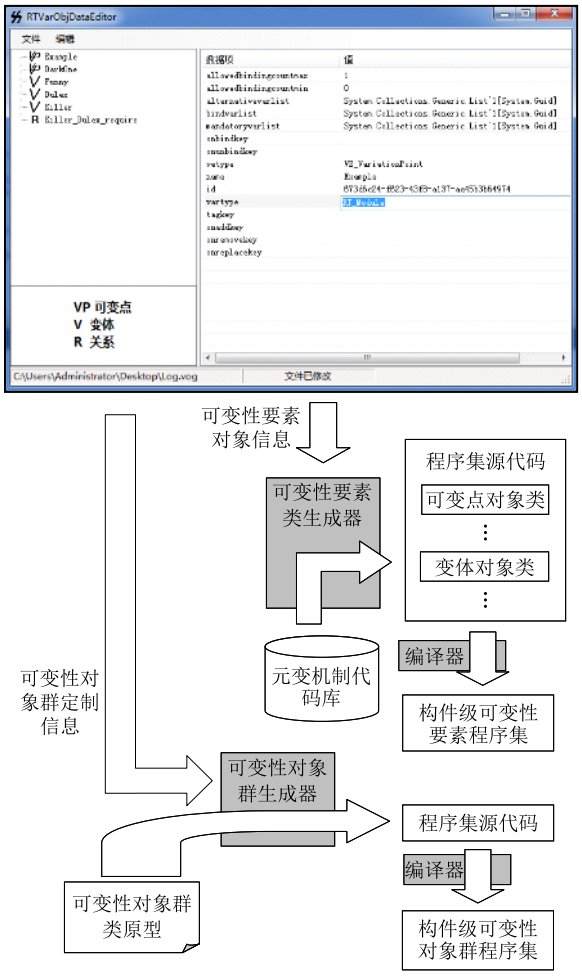


图 11　构件级可变性要素程序集和可变性对象群的生成

体系结构级可变性对象群的生成与构件级可变性对象群类似.但不同的是,其所在的程序集在编译时不需要用户描述其信息.这是因为其中只包含一个可变性对象群、一个可变点和一个变体(软件最初的体系结构配置).该变体是系统在初始化时利用.Net Framework的反射机制收集软件当前配置信息生成的.

4　实例研究及性能测试

本节以个人云为实验背景,以实际应用案例展示如何根据需求变化来调整软件系统的运行时可变性,进而展示了元变机制对软件运行时可变性动态演化的支持.而后测试比较了演化过程中对不同实现形式可变性要素的不同操作的时间开销,以展示元变机制的高效性.

4.1　案例描述

个人数据资源网络(Personal Data Resource Network,PDRN)是一个部署于一组互联的个人设备(如 PC,移动终端等)之上,用于管理数据资源(如数字视频)的个人云(Personal Cloud)①,旨在向单个用户或单个家庭提供随时随地的可定制的服务,例如分布式数据存储、基于语义网的资源自动分类等.

PDRN 的一项功能性需求是资源共享,即按照用户的意愿向其他人或公共网络开放云中管理的一部分资源供其访问.这种共享的方式按照资源的内容、开放的范围等而有所不同.这种差异到了软件的运行阶段即表现为运行时可变性,反映了设计者对于资源共享这一功能性需求的理解和假设.

我们以资源清理为例演示如何根据需求变化来演化软件系统的运行时可变性:基于安全性考虑,PDRN 中将用户指定的共享资源从云中的其他节点复制到一个长期在线的节点(称共享节点)上.在具有一定权限的情况下,访问者可以浏览或下载该节点上公开的资源内容.由于共享节点的存储空间是有限的,共享节点会依照用户动态指定的策略自动删除满足一定条件(如有效期已过)的资源,以为新的共享资源留出空间.如图 12 所示,为尽量多地保留存储空间,系统可以优先删除过了一定日期(日期阈值,为可变点,由该策略描述;这意味着对于所有资源,日期阈值都相同)的较大的资源;而为尽快保留存储空间,系统可以优先删除过了指定日期(有效期,为可变点,由资源的属性描述;这意味着对于每一个资源,有效期可能是不一样的)的资源.

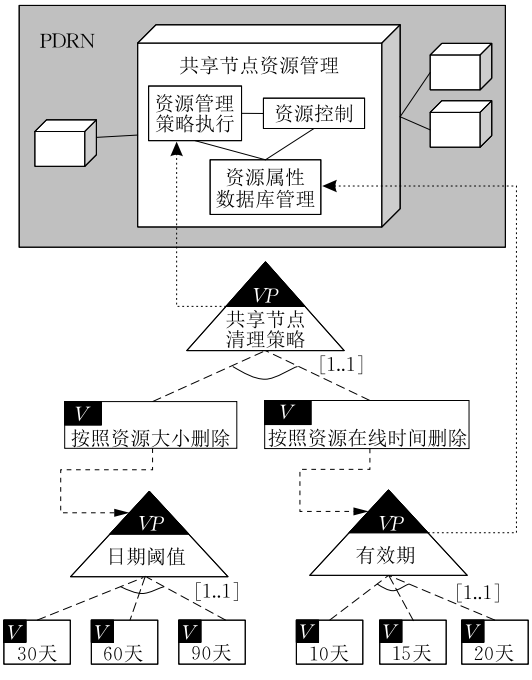


图 12　关于 PDRN 中资源共享节点清理策略的运行时可变性简化模型

① http://personal-clouds.org/wiki/Main_Page

图 12 中“共享节点清理策略”是一个功能型可变点.表示该可变点的软件制品是一个委托,位于“共享节点资源管理”构件中“过期资源检测”这一模块.其变体(即具体的策略)的实现是具有签名 bool IsOutofDate(UInt32 index)的函数.其中参数表示资源列表(位于“资源属性数据库管理”模块)中包含资源信息的数据项的索引;返回值表示依据函数内置的判断逻辑该资源是否过期.在这样的实现背景下,该可变点绑定变体的方式即将具有上述签名的函数赋予该委托.

(1)场景 1.添加优化策略(增加功能性变体)

在实际的系统运行中,设计者发现,用户希望有些资源存在于共享节点的时间要长,不能仅单凭有效期或大小删除它们以节省存储空间.换言之,用户希望增加更为优化的策略(变体),即综合考虑资源大小和在线时间的策略以供选择.

(2)场景 2.不统一的资源清理策略(增加参数型可变点和功能性变体)

在以上论述中,清理策略对于共享节点上的资源而言是统一的.然而为了更为灵活的管理资源,用户希望对不同的资源指定不同的清理策略.为解决这一问题,设计者决定将原先节点内统一的清理策略改为资源的附加属性(如同有效期一样,也是可变点),这样就可以为每个资源选择不同的变体.

4.2 运行时可变性动态演化

PDRN 中“共享节点资源管理”模块是在 .Net Framework 的基础上由多个构件实现的.采用 3.3.3 节论述的方法生成其元变机制基础设施,其中与 4.1 节问题相关的包括一个构件级可变性对象群类 ResAdminVarObjectGroup、相应的可变性要素对象类、与可变性对象群交互的用户界面等(它们在同一个程序集内),它们与“资源管理策略执行”构件 ResPolicyExecutionComponent、“资源控制”构件 ResControlComponent、“资源属性数据库管理”构件 ResPropertyDataSetComponent 等关联.同时,共享节点还存在一个体系结构级的可变性对象群 ShareNodeVarObjectGroup.

为支持动态演化,使用可变性对象来表示两个场景中所涉及的可变点和变体.以“共享节点清理策略”可变点及相关变体为例,其相关简要代码如图 13 所示.

```
public delegate bool IsOutofDate(UInt32 index);
public class RTVariationPoint_ClearPolicy: RTVariationPoint/*“共享节点清理策略”可变点对象类*/
{
    public RTVariationPoint_ClearPolicy()
    {
        this.AssociatedVariantList = new List< RTVariant_ClearPolicy >();
        this.AssociatedVariantList.Add(new RTVariant_ClearPolicy(“按照资源大小删除”, IsOutofDate_BySize));//关联变体
        this.AssociatedVariantList.Add(new RTVariant_ClearPolicy(“按照资源在线时间删除”, IsOutofDate_BySize));
        this.VarType = RT_Functionality;/*说明是功能性可变点*/
        ……
    }
    internal IsOutofDate VPEntity; // ResPolicyExecutionComponent构件会调用这个委托
    public bool BindVariant(UInt32 index)
    {
        this.BoundVariantList.Add(this.AssociatedVariantList[index]);//绑定变体
        ……
    }
    ……
}
public class RTVariant_ClearPolicy: RTVariant/*相关变体对象类*/
{
    public RTVariant_ClearPolicy(string name, IsOutofDate v)
    {
        this.Name = name;
        this.VarType = RT_Functionality;/*这个必须与可变点保持一致*/
        this.Realizedentity = v;
    }
    private string Name;
    private IsOutofDate realizedentity;
    private VarType vartype;
    public void override OnAdd(RTVarObjectGroup vog) {}
    //由于IsOutofDate方法没有状态,所以不需要在增加该类型的变体后维护系统的一致性,此函数为空*/
    public void override OnRemove (RTVarObjectGroup vog) {}//同上
    public void override OnReplace (RTVarObjectGroup vog) {}//同上
    ……
}
```

图 13 “共享节点清理策略”可变点对象类及相关变体对象类的简要代码

在可变性对象的基础上,论述问题的解决.

(1)场景 1 问题的解决

该场景相关的可变性要素即图 12 中的“共享节点清理策略”,在“资源管理策略执行”构件中被表示.在运行时,相应的可变点对象实例化了可变点对象类的子类 ResDisposingPolicy,并管理与其关联的可变点的信息(作为该类的属性).

作为 ResDisposingPolicy 的方法,该可变点对变体的绑定由向委托赋值(函数地址)实现(如图 13 所示).绑定操作本身一般是 ResDisposingPolicy 的

实例根据用户的指令进行自我调用.“资源管理策略执行”构件本身不关心绑定问题.

待增加的优化策略具有与可变点已关联的其他变体相同的函数签名.由于它的实现不在系统已经加载的任何构件实例当中,因而可以通过增加新的实现该函数的构件或替换“资源策略库管理”构件 ResPolicySetAdminComponent 为新的实现了函数的构件.基于维护成本的考虑,本文采用第 1 种方法.

元变机制对于优化策略变体增加的过程简述为:ResAdminVarObjectGroup 接到指令增加变体(具有 *IsOutofDate* 签名的函数,对应变体对象为 OptPolicyVarObject),并根据后者的信息确定待增加的构件 NewVarComponent;而后向 ShareNodeVarObjectGroup 发送增加该构件的请求;最后 ShareNodeVarObjectGroup 满足请求.

采用 3.3.3 节的过程 1 作为一般步骤增加包含的构件的具体过程以活动图表示如图 14.其中“Assembly.LoadFile”是调用 .Net Framework 中 Assembly 的静态方法 LoadFile 生成待增加的构件所位于的程序集的实例,“CreateInstance”则是利用该实例的 CreateInstance 方法创建程序集中的构件实例和变体对象实例.实际上新增加的构件中还包含其他可变性要素,其对应的可变性要素对象也是通过上述方法创建的,但未在图 14 中描述.

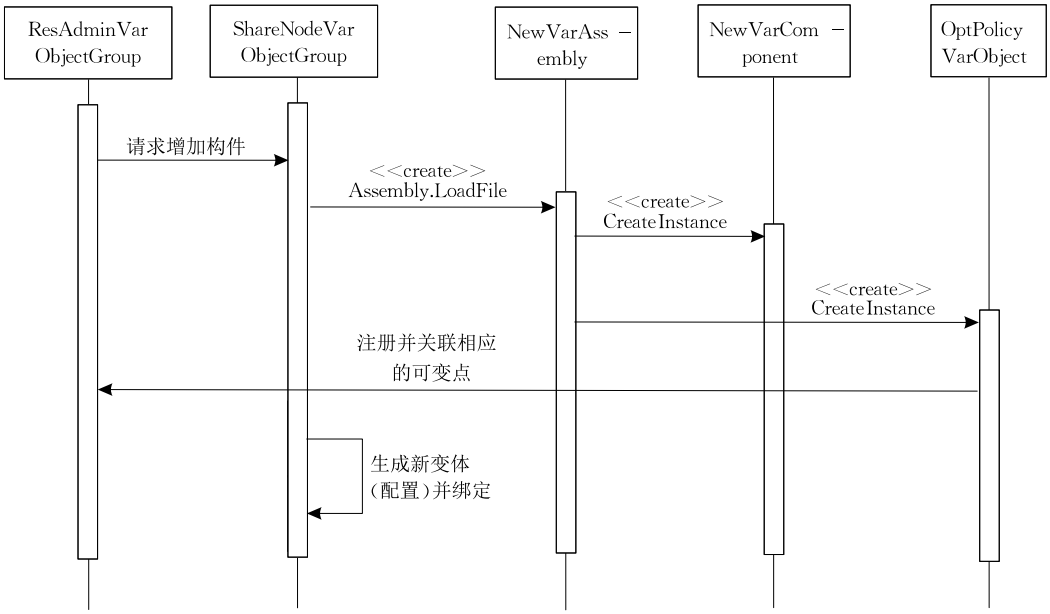


图 14　以新变体增加为目的的基于 .Net Framework 的构件增加过程

系统在元变机制基础设施中提供了能够与可变性对象群交互的用户界面(控制面板(Control Panel)和监视器(Monitor)),可以观察到上述元变机制执行的过程前后可变性模型的变化.如图 15 所示,左侧矩形框内标定的是演化之前“共享节点清理策略”的可变点,在该处只有两个变体可供选择,而右侧圆角矩形框内标定的是演化之后的同名可变点,可以发现优化策略变体已经加入.同时也可以观察到演化过程增加了“过期权重”可变点.它是指优化策略中过期时间在综合指数中占的权重,不属于场景 1 中所要满足的需求的范围,但它是优化策略变体所依赖(Require)的可变点,位于实现新变体的构件,所以也伴随着构件增加被加入系统中,受到 ResAdminVarObjectGroup 的管理.

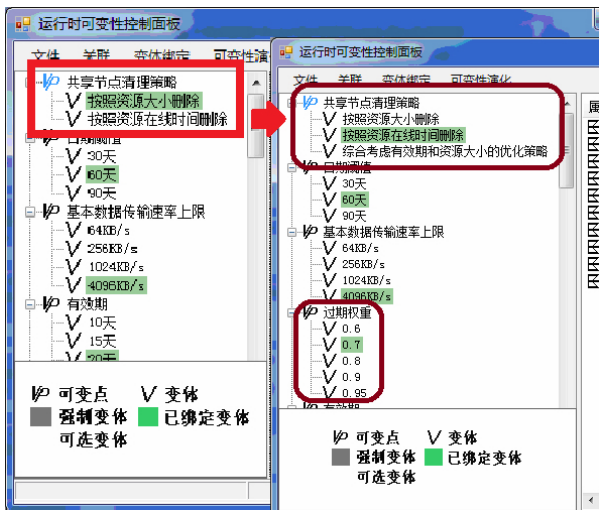


图 15　演化前后的可变性要素数量的变化

为了显示场景 1 可变性演化的有效性,我们通过控制面板在“共享节点清理策略”处选择绑定新增加的优化策略.我们设置“有效期”为 10 s,“过期权重”可变点绑定 0.6(即 ResPolicyExecutionComponent

在清理资源时“有效期”占 0.6 的权重,“大小”占 0.4 的权重).共享节点上原有 A,B,C 三份资源,表 2 显示了系统在做出如上设置的 10 min 后对这些资源的清理情况.

在资源 A,B,C 具有表 2 中大小、有效期和超过有效期时间的情况下:如果按照资源大小清除的策略,清理的先后顺序应当是 B→C→A;如果是按在线时间清除的策略(超过有效期时间加上有效期即在线时间),清理的先后顺序应当是 A→C→B;而从表 2 可以看出,资源 A 是最先被清除的,其次是资源 B,而后是 C.这说明在绑定优化策略后,共享节点对资源的清理行为发生了改变.应当指出的是,系统之所以可以能够这样绑定是因为我们之前增加优化策略变体的演化过程得到了有效完成.

表 2 “共享节点清理策略”在绑定优化策略 10 min 后共享节点对资源的清理情况

	大小	有效期	超过有效期时间	清除时间
资源 A	10 KB	10 s	43.2 min	15 点 34 分 58 秒 699 毫秒
资源 B	509 MB	10 s	9.9 min	15 点 36 分 13 秒 613 毫秒
资源 C	315 KB	10 s	15.0 min	15 点 41 分 43 秒 371 毫秒

(2)场景 2 问题的解决

从可变性角度而言,满足场景 2 的需求需要做到两点:① 增加可变点.在描述资源的元数据(如名称、类型、大小、有效期等资源附加属性)中增加“定制清理策略”可变点(参数型,从元数据的角度来看是附加属性),这意味着每一个资源的信息描述中都会包含这一可变点;② 增加变体.在图 12 所示的“共享节点清理策略”可变点增加一个功能型变体“个性策略”(也是具有 *IsOutofDate* 签名的函数),其在运行时会执行资源的“定制清理策略”可变点具体指定的清理策略.

可变点通常涉及软件的深层设计,以预定义的方式被置于系统中[1,3].设计者最初并未考虑到个性化的清除策略需求,所以未在相关构件 ResPropertyDataSetComponent 中设计出具有相应可变点的资源元数据.设计者也未考虑到在需求演化时元数据的变化,所以未将其设计为数据集型可变点,否则解决场景 2 中的问题会相对简单一些.

基于上述讨论,增加可变点的过程具体如下.

过程 4. 清理策略可变点的增加.

1. ResAdminVarObjectGroup 接到用户指令增加可变点;

2. ResAdminVarObjectGroup 根据新增加的可变点和关联变体的基本信息和人为的通过控制面板的设置,确定其实现所在的构件 ResPropertyDataSetComponent_Ver_02;

3. 向 ShareNodeVarObjectGroup 发送消息,请求替换 ResPropertyDataSetComponent 为该构件;

4. ShareNodeVarObjectGroup 满足请求.

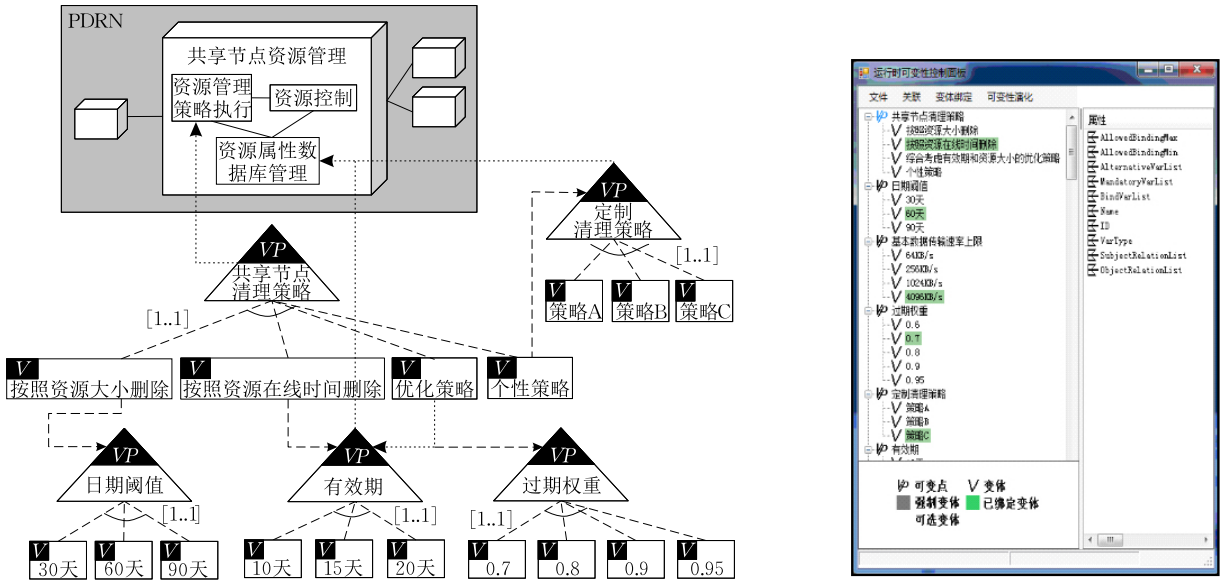
本文中替换构件以过程 2 为参考,除第 5 步状态迁移之外其余的步骤或与场景 1 的类似,或实现相对简单,本节不再赘述.

状态迁移的过程包括新增加可变点的状态初始化、原有可变性要素的状态迁移以及构件本身其他状态的迁移.因为新增的可变点“定制清理策略”绑定的变体数量下限和上限都是 1,所以其至少应当关联并绑定一个变体才能够是有效的.该可变点在初始化时关联变体“策略 A”、“策略 B”、“策略 C”,并默认绑定第一个变体.原构件内可变性要素的状态迁移主要是将可变点与变体之间的绑定关系转化为新构件中的绑定关系.人们需要保证两者之间的一致性,以使新构件能够保持人们对构件内实现的可变性的设定,并能够提供原构件的服务.为了使软件能够持续运行,除了可变性之外,构件本身的其他状态也需要在新旧构件实例之间迁移.

可以采用我们前期工作[37]的方式来完成上述状态迁移过程.这就要求旧构件将自身的状态(如内存变量、可变性绑定信息)暴露给 ShareNodeVarObjectGroup,由其提取到缓冲区内而后再根据系统维护者写的状态迁移脚本将之转化为新构件的状态.

经过上述过程的演化之后,PDRN 中关于资源共享节点清理策略的运行时可变性模型改变为如图 16(a)所示.在此上下文中,如果用户为每个资源定制个性化的清理策略,则需要在“共享节点清理策略”处选择“个性策略”变体,而后才能在“资源属性数据库管理”构件的支持下为每一个资源指定不同的清理策略 ID(参数).而这些清理策略 ID 对应着判断资源是否过期的功能性代码,在个性策略对应的 *IsOutofDate* 函数运行时被调用.图 16(b)为经过两个场景演化之后共享节点运行时可变性模型.

图 16 显示,“共享节点清理策略”又增加了一个新变体“个性策略”,它依赖新增的可变点“定制清理策略”.后者在共享节点的每个资源的元数据描述中都具有一个实例.换言之,用户可以为每个资源选择属于它自身的清理策略,而 ResPolicyExecutionComponent 在使用“个性策略”时会考察这些策略并执行之.



(a) 经过场景1和2演化后的关于PDRN中资源共享节点清理策略的运行时可变性简化模型　　(b) 通过监视器观察到的资源共享节点的运行时可变性要素

图　16

我们使用场景 1 中的资源 A,B,C 来说明上述演化过程的有效性. 3 份资源仍然具有表 2 中的大小、有效期和超过有效期时间,其分别在“定制清理策略”处绑定策略 A,B,C,在做出如上设置 10 min 后,清理情况如表 3 所示.

表 3　“共享节点清理策略”在绑定个性策略 10 min 后共享节点对资源的清理情况

	定制清理策略	清除时间
资源 A	策略 A(按照资源被下载次数决定是否清理)	16 点 12 分 23 秒 192 毫秒
资源 B	策略 B(按照在线时间长短决定是否清理)	16 点 10 分 07 秒 133 毫秒
资源 C	策略 C(资源评论星级决定是否清理)	16 点 13 分 52 秒 831 毫秒

可以看出,资源清理的先后顺序是 B→A→C,与场景 1 内所述情况不同. 这说明“共享节点清理策略”在绑定“个性策略”之后,系统行为又发生了变化,场景 2 的需求得到了满足. 这进一步说明上述的运行时可变性动态演化使得清理策略对不同资源的个性化成为可能,也说明了演化过程的有效性.

4.3　性能测试

我们在 Windows 7 专业版.Net Framework 4.5 环境下,利用 Visual Studio 2010 的性能分析器,在上述个人云共享节点上下文中,针对不同实现形式的可变性要素的增加、删除、替换操作的执行时间销进行了测试和比较. 测试是为了展示采用 3.3.3 节生成技术构建的软件系统的运行时可变性演化的执行效率,因而不再说明演化的具体目的(即为了应对怎样的需求或环境变化).

我们按可变性要素的不同实现形式将演化操作分为六组进行实验,我们按增加可变点→增加变体→替换变体→删除变体→替换可变点→删除可变点的顺序分别对每组相应的操作进行 100 万次调用(分批 10 次运行测试,每批次测试循环调用 10 万次)并记录时间开销(包括可能的体系结构动态调整的时间和状态迁移的时间). 其中参数型可变性要素实现采用 32 位整数类型变量,功能型可变性要素实现采用上述场景 1 中优化策略的 *IsOutofDate* 方法,构件型可变性要素实现采用实现该方法的程序集,数据集型可变性要素实现采用上述描述资源属性的元数据类型,工作流型可变性要素实现采用描述清理策略的数据,配置型可变性要素实现采用描述(实现清理策略的)构件的数据.

附录 B 展示了 6 组实验中演化操作过程 10 次测量的平均时间开销(这里平均值代表了 10 万次循环调用耗时的总和). 实验数据表明元变机制是较为高效的. 按每次操作所使用的总时间计算,平均一次增加、删除、替换可变性要素耗时大约 0.0177 ms.

同时可以观察到,不论实现形式和操作类型,一项操作的各个步骤的时间开销的加和小于相应的总时间,这是因为.Net Framework 会在可变性演化时根据程序代码自动完成相关的系统调用(如内存垃圾回收),导致演化耗时比仅执行代码时间(称应用程序非独占时间[35])要长.

从一次可变性演化过程耗时的角度考虑,除去

不需要调整体系结构的删除操作之外,体系结构调整(主要是增加或替换构件)耗时占总开销的 86.43%～88.69%.而其他步骤如调整可变性对象群(将新可变性要素对象在其中注册)等所消耗的时间不超过10%.体系结构作何调整取决于待演化的可变性要素与演化目的的关系,而与其实现形式基本无关.

如果除去体系结构调整耗时和系统调用时间,不同实现形式的可变性要素的操作时间开销是较为接近的(标准差在 0.2692～1.8831 之间),如图 17 所示.无论何种实现形式,替换可变性要素的时间开销大约是增加该种要素和删除相应的时间开销的总和.这也符合我们在 3.2 节的论述.可以认为,在不考虑体系结构调整的情况下,我们所提出的支持运行时可变性演化的方法是与相应的可变性要素实现形式无关的,达到了引言中提出的目标.

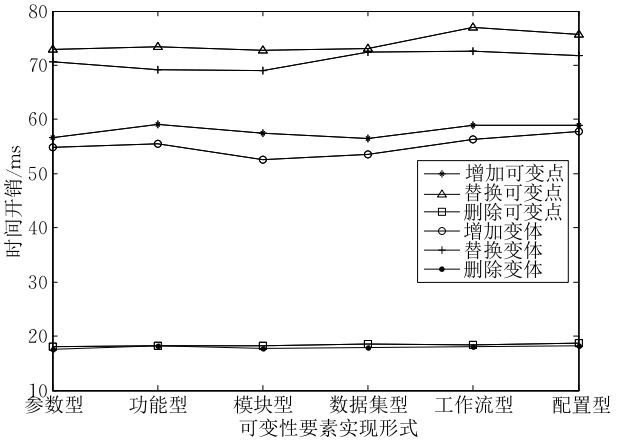


图 17 不同实现形式的要变性要素演化操作的时间开销比较(不考虑体系结构调整)

5 结 论

软件运行时可变性近年受到来自适应系统、动态软件产品线等软件工程领域的关注.但在大多数情况下,自适应系统或类似的系统难以满足在运行期持续变化的(而且往往是超过预期的)需求和环境要求.本文指出造成该问题的主要原因是它们对静态运行时可变性模型的依赖.运行时可变性的动态演化则是解决问题的关键.针对该领域内缺乏对实现技术的研究这一问题,本文提出了元变机制,其可通过可变性对象对运行时可变性模型进行表示,并通过体系结构动态调整实现对可变性要素数量、属性或关系的改变.可变性对象在抽象层面上屏蔽了可变性要素的具体实现和可变性演化的底层细节,而使人们专注于可变性模型及其变化.后者在两层可变性对象群的主导下利用构件的增加、替换和删除具体实现了可变性演化的主要过程.本文还论述了支持元变机制运作的基础设施,以及辅助开发者生成这些基础设施的技术.

下一步的工作主要围绕体系结构调整和可变性要素的增加、替换和删除展开:从性能测试结果可以看出,相对可变性演化而言,体系结构的动态调整带来的代价仍然较高.因而我们将在考虑可变性要素异构性、可追踪性和抽象性的同时,寻求一种更为轻量级的方法达到运行时可变性演化的目的.未来工作还包括对元变机制基础设施的完善,加入模型一致检查机制及设施,并在更广泛的场景下进行实验验证.

参 考 文 献

[1] Svahnberg M, Gurp J, Bosch J. A taxonomy of variability realization techniques: Research articles. Software: Practice and Experience, 2005, 35(8): 705-754

[2] Pohl K, Böckle G, Linden F. Software product line engineering: Foundations, principles, and techniques. Berlin Heidelberg: Springer-Verlag, 2005

[3] Galster M, Weyns D, Tofan D, et al. Variability in software systems: A systematic literature review. IEEE Transactions on Software Engineering, 2014, 40(3): 282-306

[4] Salehie M, Tahvildari L. Self-adaptive software: Landscape and research challenges. ACM Transactions on Autonomous and Adaptive Systems, 2009, 4(2): 1-40

[5] Hallsteinsen S, Hinchey M, Park S, Schmid K. Dynamic software product lines. IEEE Computer, 2008, 41(4): 93-95

[6] Bosch J. Design & Use of Software Architectures: Adopting and Evolving a Product Line Approach. Boston: Addison-Wesley, 2000

[7] Bencomo N, Lee J, Hallsteinsen S. How dynamic is your dynamic software product line?//Proceedings of the 14th International Software Product Line Conference (SPLC 2011). Munich, Germany, 2010: 61-68

[8] Esfahani N, Elkhodary A, Malek S. A learning-based framework for engineering feature-oriented self-adaptive software systems. IEEE Transactions on Software Engineering, 2013, 39(11): 1467-1493

[9] Mosincat A, Binder W, Jazayeri M. Achieving runtime adaptability through automated model evolution and variant selection. Enterprise Information Systems, 2014, 8(1): 67-83

[10] van Deursen A, de Jonge M, Kuipers T. Feature-based product line instantiation using source-level packages//Proceedings of the International Software Product Lines Conference (SPLC 02). San Diego, USA, 2002: 19-30

[11] Ye H, Liu H. Approach to modeling feature variability and dependencies in software product lines. IEE Proceedings of Software, 2005, 152(3): 101-109

[12] Loesch F, Ploedereder E. Optimization of variability in software product lines//Proceedings of the 11th International Software Product Lines Conference (SPLC 07). Kyoto, Japan, 2007: 151-162

[13] Ding Bao-Bao, Shen Li-Wei, Peng Xin, Zhao Wen-Yun. Synchronize domain models and application models of evolving software product line. Journal of Chinese Computer Systems, 2015, 36(2): 197-204(in Chinese)
(丁宝宝, 沈立炜, 彭鑫, 赵文耘. 软件产品线领域模型与应用模型的通用同步演化方法. 小型微型计算机系统, 2015, 36(2): 197-204)

[14] Deng Han-Bing, Zhao Li-Jun, Zhang Xia, Liu Ji-Ren. A method for analyzing the evolution of a concept lattice based software product line. Journal of Northeastern University (Natural Science), 2011, 32(3): 331-339(in Chinese)
(邓寒冰, 赵立军, 张霞, 刘积仁. 一种基于概念格的软件产品线演化分析方法. 东北大学学报(自然科学版), 2011, 32(3): 331-339)

[15] Nie Kun-Ming, Zhang Li. Software architecture variability modeling method for software intensive system. Journal of Frontiers of Computer Science and Technology, 2014, 8(7): 823-835(in Chinese)
(聂坤明, 张莉. 面向软件密集型系统的体系结构可变性建模. 计算机科学与探索, 2014, 8(7): 823-835)

[16] Ali Babar M, Chen Lian-Ping, Shull F. Managing variability in software product lines. IEEE Software, 2010, 27(3): 89-91

[17] Dobson S, Denazis S, Fernandez A, et al. A survey of autonomic communications. ACM Transactions on Autonomous and Adaptive Systems, 2006, 1(2): 223-259

[18] Ding Bo, Wang Huai-Min, Shi Dian-Xi, Li Xiao. Component model supporting trustworthiness-oriented software evolution. Journal of Software, 2011, 22(1): 17-27(in Chinese)
(丁博, 王怀民, 史殿习, 李骁. 一种支持软件可信演化的构件模型. 软件学报, 2011, 22(1): 17-27)

[19] Kephart J O, Chess D M. The vision of autonomic computing. IEEE Computer, 2003, 36(1): 41-52

[20] Huebscher M C, McCann J A. A survey of autonomic computing: Degrees, models, and applications. ACM Computing Surveys, 2008, 40(3): Article 7

[21] Northrop L M, Clements P. A Framework for Software Product Line Practice (Version 5.0). Pittsburgh, USA: Software Engineering Institute, 2007

[22] Bencomo N, Hallsteinsen S, de Almeida E S. A view of the dynamic software product line landscape. IEEE Computer, 2012, 45(10): 36-41

[23] Hallsteinsen S, Hinchey M, Park S, Schmid K. Dynamic software product lines. IEEE Computer, 2008, 41(4): 93-95

[24] Villela K, Silva A, Vale T, de Almeida E S. A survey on software variability management approaches//Proceedings of the 18th International Software Product Line Conference (SPLC 2014). Florence, Italy, 2014: 151

[25] Gacek C, Anastasopoules M. Implementing product line variabilities. ACM SIGSOFT Software Engineering Notes, 2001, 26(3): 109-117

[26] Amin F, Mahmood A K, Oxley A. An analysis of object oriented variability implementation mechanisms. ACM SIGSOFT Software Engineering Notes, 2011, 36(1): 1-4

[27] Bachmann F, Clements P C. Variability in software product lines. Software Engineering Institute, Carnegie-Mellon University, Pittsburgh PA: Technical Report CMU/SEI-2005-TR-012, 2005

[28] Capilla R, Bosch J, Kang K. Systems and Software Variability Management: Concepts, Tools and Experiences. Berlin Heidelberg: Springer, 2013

[29] Passos L, Czarnecki K, Apel S, et al. Feature-oriented software evolution//Proceedings of the International Workshop on Variability Modeling of Software—Intensive Systems 2013. Pisa, Italy, 2013: 23-25

[30] Chen Kun, Zhang Wei, Zhao Hai-Yan, Mei Hong. An approach to constructing feature models based on requirements clustering//Proceedings of the 13th International Conference on Requirements Engineering. Paris, France, 2005: 31-40

[31] Capilla R, Bosch J, Trinidad P, et al. An overview of dynamic software product line architectures and techniques: Observations from research and industry. Journal of Systems and Software, 2014, 91: 3-23

[32] Bass L, Clements P, Kazman R. Software Architecture in Practice. New York: Addison-Wesley, 1998

[33] De Panfilis S, Berre A J. Open issues and concerns on component based software engineering//Proceedings of the 9th International Workshop on Component-Oriented Programming (WCOP 2004). Oslo, Norway, 2004: 14-18

[34] Ortin F, Redondo J M, Perez-Schofield J B G. Efficient virtual machine support of runtime structural reflection. Science of Computer Programming, 2009, 74(10): 836-860

[35] Microsoft Corporation. Microsoft C# Language Specifications. Redmond: Microsoft Press, 2010

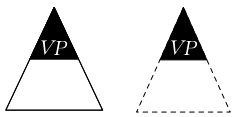
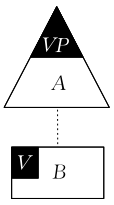
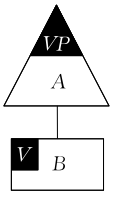
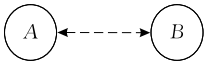
[36] Seifzadeh H, Abolhassani H, Moshkenani M S. A survey of dynamic software updating. Journal of Software: Evolution and Process, 2013, 25(5): 535-568

[37] Liu Ji-Wei, Mao Xin-Jun. Towards realization of evolvable runtime variability in Internet-based service systems via dynamical software update//Proceedings of the 6th Asia-Pacific Symposium on Internetware on Internetware (Internetware 2014). Hong Kong, China, 2014: 97-106

附录 A. 软件运行时可变性建模.

本文利用 OVM[2] 对运行时可变性进行建模. 由于原有的 OVM 不包含运行时可变性独有的要素,而且难以突出运行时可变性的动态性,本文增加了若干图元,扩展了 OVM 所能表达的语义. 扩展后的 OVM 可用于在逻辑上表达运行时可变性模型及其演化,其图元见附表 1.

附表 1 扩展后的 OVM 图元

图元名称	图形符号	基本语义
可变点		软件中可以发生变化的位置(右侧图元表示演化过程中新增加的可变点或待删除的可变点)
变体		软件发生变化时,在相关可变点做出的选择(右侧图元表示演化过程中新增加的变体或待删除的变体)
可选(可变性依赖)		可变点和与之关联的变体之间的一类关系,图例中表示可变点 A 可以选择变体 B
强制(可变性依赖)		可变点和与之关联的变体之间的一类关系,图例中表示可变点 A 必须选择变体 B
依赖(限制性依赖)		若可变性要素 A 是有效的,则可变性要素 B 必须是有效的
排斥(限制性依赖)		若可变性要素 A 是有效的,则可变性要素 B 必须是无效的
(可变点)制品依赖		可变性要素 A 实现为软件制品 B

附录 B. 元变机制演化操作时间开销测试结果.

附表 2 记录了 6 组实验中演化过程中重要步骤的时间开销. 每一项数据是相应步骤中 10 批次测量结果的平均值. 因为每批次均为对相应操作的 10 万次循环调用,因此每项数据代表的是 10 万次操作中相应步骤时间开销的总和.

附表 2 中符号说明:

t_{new}:新可变性对象生成及初始化时间;

t_{gp}:可变性对象群调整(比如注册可变性对象)时间;

t_{trans}:可变性对象状态迁移时间;

t_{arch}:体系结构调整时间;

t_{total}:操作总时间.

附表 2 参数型可变性要素增加、删除或替换过程的时间开销

(单位:ms)

	t_{new}	t_{gp}	t_{trans}	t_{arch}	t_{total}
增加可变点	32.94	8.56	15.22	2352.43	2705.67
替换可变点	33.54	23.97	15.48	2127.94	2580.81
删除可变点	0	18.05	0	0	19.62
增加变体	31.58	8.22	15.06	2517.13	2705.09
替换变体	31.10	24.09	15.42	2209.21	2538.76
删除变体	0	17.55	0	0	41.08

附表 3 功能型可变性要素增加、删除或替换过程的时间开销

(单位:ms)

	t_{new}	t_{gp}	t_{trans}	t_{arch}	t_{total}
增加可变点	35.45	8.38	15.28	2362.76	2723.78
替换可变点	33.08	24.27	16.09	2124.12	2558.71
删除可变点	0	18.26	0	0	68.19
增加变体	31.80	8.36	15.265	2317.43	2654.57
替换变体	28.64	24.51	15.98	2225.01	2536.82
删除变体	0	18.21	0	0	18.77

附表 4 模块型可变性要素增加、删除或替换过程的时间开销

(单位:ms)

	t_{new}	t_{gp}	t_{trans}	t_{arch}	t_{total}
增加可变点	33.99	8.33	15.15	2362.76	2697.61
替换可变点	33.97	24.05	14.76	2286.96	2583.33
删除可变点	0	18.22	0	0	72.41
增加变体	29.25	8.31	15.04	2213.67	2640.29
替换变体	29.34	24.26	15.40	2152.61	2513.08
删除变体	0	17.75	0	0	64.51

附表 5 数据集型可变性要素增加、删除或替换过程的时间开销

(单位:ms)

	t_{new}	t_{gp}	t_{trans}	t_{arch}	t_{total}
增加可变点	32.60	8.48	15.34	2331.32	2788.02
替换可变点	32.47	24.32	16.29	2288.30	2613.49
删除可变点	0	18.47	0	0	45.30
增加变体	28.89	8.48	16.11	2297.22	2686.24
替换变体	32.15	24.56	15.66	2199.51	2557.60
删除变体	0	17.97	0	0	62.61

附表 6 工作流型可变性要素增加、删除或替换过程的时间开销

(单位:ms)

	t_{new}	t_{gp}	t_{trans}	t_{arch}	t_{total}
增加可变点	34.72	8.47	15.79	2311.86	2731.21
替换可变点	36.44	24.32	16.25	2153.72	2578.98
删除可变点	0	18.46	0	0	19.10
增加变体	32.74	8.49	15.16	2169.90	2633.36
替换变体	32.24	24.56	15.74	2277.09	2530.45
删除变体	0	18.08	0	0	106.65

附表 7 配置型可变性要素增加、删除或替换过程的时间开销

(单位:ms)

	t_{new}	t_{gp}	t_{trans}	t_{arch}	t_{total}
增加可变点	35.20	8.61	15.07	2252.71	2730.09
替换可变点	35.11	24.68	15.97	2235.83	2556.78
删除可变点	0	18.69	0	0	19.03
增加变体	32.25	8.62	16.87	2213.78	2630.66
替换变体	31.51	24.85	15.41	2243.99	2513.28
删除变体	0	18.22	0	0	49.38

LIU Ji-Wei, born in 1987, Ph. D. candidate. His research interests include software runtime variability and its dynamic evolution, software architecture and design, self-adaptive systems.

MAO Xin-Jun, born in 1970, Ph. D., professor, Ph. D. supervisor. His research interests include software engineering, multi-agent systems, self-adaptive and self-organizing systems.

Background

The study conducted in this paper belongs to the area of software variability and that of software evolution. Software variability, especially runtime variability, draws much attention of researchers in the community of software engineering in recent years. However, they typically focus on the application of runtime variability to self-adaptive systems or other similar systems like dynamic software product lines. The evolution of runtime variability is concerned only from the perspective of logic models other than implementation.

We propose an approach to evolve runtime variability with regard to its number, attribute or relation in an implemental and semi-automatic manner. The work is part of the study on Model and Architectural Support for Evolution of Runtime Variability, which aims to make distributed systems like personal clouds continuously meet varying requirements. Our previous work is related to evolving runtime variability on the basis of dynamic software updating, which focuses on the techniques of replacement of entire process and state transformation. The concept of evolvable runtime variability is proposed in the relevant paper (i. e., Ref. [37]) published in Internetware 2014.

The work is financially supported by the National Natural Science Foundation of China under Grant Nos. 61379051, 61532004, and the Program for New Century Excellent Talents in University under Grant No. NCET-10-0898.