中国科学: 信息科学 2020 年 第 50 卷 第 11 期: 1595–1611
SCIENTIA SINICA Informationis
《中国科学》杂志社
SCIENCE CHINA PRESS
可成长软件专题 · 评述

可成长软件理论方法和实现技术: 从范型到跨越

许畅[1,2*], 秦逸[1,2], 余萍[1,2], 曹春[1,2], 吕建[1,2]

1. 南京大学计算机软件新技术国家重点实验室, 南京 210023
2. 南京大学计算机科学与技术系, 南京 210023
* 通信作者. E-mail: changxu@nju.edu.cn

收稿日期: 2020–04–01 ; 修回日期: 2020–07–07 ; 接受日期: 2020–09–30; 网络出版日期: 2020–11–10

国家重点研发计划 (批准号: 2017YFB1001801) 和国家自然科学基金 (批准号: 61932021, 61902173) 资助项目

摘要 在云计算和大数据的技术背景下, “人 – 机 – 物” 三元融合的应用模式正不断加速社会的信息化进程, 并对软件系统的自适应和持续演化能力提出了新的需求. 本文探索了面临软硬件环境及外部资源不断变迁挑战下的可成长网构软件理论方法和实现技术, 从软件可成长性问题的由来, 至可成长性概念的内涵和可成长软件的范型机理, 在开放环境感知与自适应、无缝演化和过程演进, 以及演化质量评估方法和保障机制 3 个方面系统分析了可成长软件的技术挑战并介绍了当前的技术进展, 以支撑软件系统在不断成长视角下的长期生存.

关键词 可成长软件, 范型机理, 自主适应, 持续演化

1 引言

云计算, 大数据技术和 “人 – 机 – 物” 三元融合的应用模式正不断加速社会的信息化进程, 在此背景下, 软件作为信息社会的基础设施, 常面临着其软硬件环境及外部资源不断变迁的挑战. 对此, 以基础软件方法学及相应关键技术的系统创新, 增强开放网络环境下软件系统的自适应和持续演化能力, 使其能够长期生存并不断成长, 成为了当前国际学术研究的新焦点. 本文以科技部国家重点研发计划 “云计算和大数据” 专项 “可持续演化的智能化软件理论, 方法和技术” 的工作为基础, 介绍我们在以智能化为手段, 以可成长为本质的可持续演化软件理论、方法和技术方面的研究进展, 以抛砖引玉, 吸引专家见解和开发者实践.

本文组织如下, 首先, 我们介绍软件的可成长性, 分析可成长性问题的由来, 以及可成长性概念的内涵和范畴; 其次, 我们探索以网构软件的思想来构建可成长软件的理论方法, 分析实现可成长网构软件的可能途径、其范型机理, 以及技术挑战; 再次, 我们讨论支撑软件可成长性的关键技术, 包括赋能关键技术和外围关键支撑, 其涉及软件开放环境感知与自适应、无缝演化和过程演进, 以及演化质

引用格式: 许畅, 秦逸, 余萍, 等. 可成长软件理论方法和实现技术: 从范型到跨越. 中国科学: 信息科学, 2020, 50: 1595–1611, doi: 10.1360/SSI-2020-0079
Xu C, Qin Y, Yu P, et al. Theories and techniques for growing software: paradigm and beyond (in Chinese). Sci Sin Inform, 2020, 50: 1595–1611, doi: 10.1360/SSI-2020-0079

量评估方法和保障机制等重要方面; 最后, 我们介绍可成长软件理论方法和技术的集成应用, 并展望未来和总结全文.

2 软件的可成长性

本节介绍软件的可成长性, 分析可成长性问题的由来, 以及可成长性概念的内涵和范畴.

2.1 软件可成长性问题的由来

从以机械化为代表的第一次工业革命, 到以电气化为代表的第二次工业革命, 一直到近代以自动化为代表的第三次工业革命, 以及当前和未来以智能化为代表的第四次工业革命, 软件已逐步成为支撑我们现代化信息社会的基础设施. 其中一个重要原因就是, 从第三次工业革命开始出现了电子计算机, 人类的计算能力和工作效率由此大幅提升, 计算机软件也因而进入了蓬勃发展的阶段. 对此, C++语言发明人 Bjarne Stroustrup 甚至高度评价: "人类文明运行在软件之上". 近年来, 云计算、大数据技术和 "人 – 机 – 物" 三元融合的应用模式正不断加速现代社会的信息化进程, 与此相应的, 软件也变得与人类生活越来越息息相关, 越来越重要.

但随之而来的, 是软件的构建和维护任务日益复杂, 不断挑战软件开发者管理和技术的能力界限. 比如说, 根据波音和空客的实证数据 [1, 2], 其软件代码可达三千万行, 并且随着其业务复杂度每增长 25%, 软件复杂度可相应增长 100%. 即使软件开发者能顶住巨大压力从事如此复杂的编码工作, 也不得不耗损自身健康作为代价. 而另一方面, 随着社会的信息化进程加速发展, 当前许多软件基础设施面临着环境、资源和需求的不断变迁, 急需重新构建或繁重维护, 这进一步加剧了问题的严重性.

在这种情况下, 软件的长期生存或单纯是生存都变得非常困难. 据悉, 有苦撑 12 年, 具有 600 多万行代码和五万多个类的烂尾开发项目, 到后期软件结构已变得异常复杂, 即使软件开发者付出极大努力也无法充分理解, 最终不得不在多次追加开发资金之后仍然垮掉[1)]. 而因为代码难以理解和注释不规范问题, 软件开发者之间也曾发生过矛盾, 甚至危及生命的事件, 使得软件的构建和维护难以持续[2)]. 即使是国际著名 Oracle 公司的成熟数据库产品, 开发人员也对其生存和维护问题怨声载道: "…… 你做不到在不破坏成千上万个现有测试的情况下更改产品中的单一行代码 …… 代码中充斥着各种各样的垃圾内容 …… 顺利的话, 会有大约 100 个失败的测试, 倒霉的话, 会有大约 1000 个失败的测试 …… 再添加几个标志, 试图解决问题 ……" [3)]. 对这样的问题, 美国国防部高级研究计划局 (Defense Advanced Research Projects Agency, DARPA) 已经开始规划, 期望设计和开发一套资源自适应软件系统 (BRASS), 它不仅能够经受一个多世纪的考验, 而且还能自行对所在生态系统中的变化做出安全、动态式的响应[4)], 从而彻底解决软件的长期生存难题.

我们认为, 软件构建和维护任务的复杂性, 实质上反映了软件的可成长性问题. 软件难以继续构建和维护, 本质上暗示了软件自身的可成长性比较差甚至缺失. 对此, 科研人员和软件开发者近几十年来一直在与软件的可成长性问题进行抗争, 并且已经付出大量管理和技术上的努力.

比如说, 在管理方面, 教科书级经典著作《人月神话》[3, 4] 很早就提出了软件危机的两点具体表现. 一是软件开发活动不具有可扩展性, 难以驾驭大规模系统, 二是团队规模和协作效率 (或沟通成本) 之间具有两难问题, 换言之, 软件的复杂性无法简单通过增加人 (即人力成本) 和月 (即时间成本)

1) https://zhuanlan.zhihu.com/p/38973085/.
2) http://finance.sina.com.cn/stock/usstock/c/2018-09-23/doc-ifxeuwwr7514854.shtml.
3) https://www.oschina.net/news/101928/about-oracle-database-code.
4) https://tech.huanqiu.com/article/9CaKrnJJO38.

来解决. 此外, 为缓解软件危机, 极限编程 XP [5] 提出了系列措施来提高软件开发的效率和增进开发人员对彼此代码的理解, 如使用用户故事把客户加入到需求分析的圈子当中, 或使用结对编程实现增进代码理解的编程分享等.

又比如说, 在技术方面, DevOps [6] 的提出倡导了软件开发模型从经典的瀑布模型, 到后来的敏捷和精益开发, 以至更具效率的持续集成、持续交付、持续部署和持续运维. 此外, 设计模式 [7] 强调了复用已有的成功软件开发设计, 以提升开发效率和降低维护难度, 而代码重构 [8] 则指引了如何在不破坏原有功能的前提下, 为已有软件引入设计模式. 针对设计模式和代码重构的引入, 科研人员也深入分析其实际效用, 如香港科技大学张成志教授团队通过设计基于实际软件开发者的对比实验, 研究了雇佣具有更多工作经验的开发者和花费代价通过代码重构为已有软件引入设计模式这两种方法哪个对软件的持续维护工作更具有效用 [9], 以及软件开发者是否真能有效利用预先在软件中部署的设计模式 [10] 等问题.

如此看来, 类似于软件可成长性问题的关注一直存在, 为何现在再次讨论并探索其可能的解决方案呢? 有两点主要的原因, 一是现代信息社会对软件的构建和维护要求更加迫切了, 如物联网、软件定义和工业 4.0 等概念和技术的提出, 促使软件系统加速进入 “人 – 机 – 物” 三元融合的应用模式, 并不断承受环境、资源和需求持续变迁的压力; 二是现代计算基础设施更加成熟和有能力, 如云计算、大数据和深度学习等技术的提出, 使得原先因为计算能力不足而无法大规模开展的工作如今能够有效尝试. 在这样的前提下, 我们希望深入理解软件的可成长性问题, 并从软件自身的构架角度 (内部) 和对开发者支持的角度 (外部) 共同探讨软件可成长性的解决方案.

需要说明的是, 反映出软件可成长性问题的软件构建和维护任务的复杂性, 既有来自开发角度的原因, 也有运行态的原因. 从开发角度来说, 在软件部署前由于初始应用需求日益复杂, 而在软件部署后则因为应用需求持续变迁, 所以开发角度这方面主要是指用户的应用需求导致软件构建和维护的复杂性. 另一方面, 从运行态来说, 软件的环境和资源持续变迁, 也可能使应用难以保持原有的服务质量和内容, 而要应对这一问题, 也会产生对软件持续维护, 甚至再次构建的需求, 所以运行态这方面主要是指环境和资源变迁所导致的软件构建和维护的复杂性. 这两方面综合起来, 我们说, 软件构建和维护的复杂性即可成长性难题, 既涉及开发角度, 又有运行态的原因, 是环境、资源和需求多个来源的持续变迁所导致的, 这是一个深刻且影响广泛, 需要认真对待的问题.

2.2 软件可成长性概念的内涵和范畴

基于以上的分析, 我们定义软件可成长性概念的内涵为: 在其所处软硬件环境, 所依赖外部资源, 以及所服务用户目标不断变化的条件下, 能通过主动的感知、智能的适应和持续的重构演化, 实现长期生存并不断优化. 这一概念内涵, 界定了引发软件可成长性目标的由来 (即环境、资源和需求的变化), 也指明了达成软件可成长性的方法 (即基于感知、适应和演化的架构).

同时我们定义软件可成长性概念的范畴为: 软件成长不是 “开创天地, 从无到有” 的神话, 即软件无需开发者, 可完全自行演化出所需功能, 也可完全自行修正自身缺陷 (从任意规约自动生成程序的问题是不可判定的 [11], 更不用提无规约情况下的程序自动生成问题); 相反, 软件成长是一个 “不断努力, 持续演进” 的过程, 即软件在自身架构上需要明确考虑其适应和演化问题, 支持软件适应变化和版本演化, 使其生存时间更长, 并对开发者支持使其更容易实现软件新增功能和修复缺陷.

3 可成长网构软件理论方法初探

本节探索以网构软件的思想来构建可成长软件的理论方法, 分析实现可成长网构软件的可能途径、其范型机理, 以及技术挑战.

3.1 网构软件理论方法发展的新阶段

网构软件 (Internetware) 理论方法 [12,13] 已经发展了十多年, 从一开始其目标就设立为: 让软件从封闭、静态和可控的传统环境走向开放、动态和难控的互联网环境, 这也暗示着软件规约从封闭走向开放、包容环境的不确定性. 由此, 网构软件的目标在本质上已包括了当下复杂软件系统正不断承受环境、资源和需求持续变迁的挑战, 因而网构软件理论方法从一开始就有相当的预见性. 基于这样的预见, 我们期望软件变得更智能, 转变自身从 “以不变应万变” 的消极姿态变成具有 “随机应变” 的主动能力, 以应对当下环境、资源和需求不断变迁的挑战.

网构软件理论方法的研究经历了前后三期科技部 973 项目, 循序渐进, 分别探索了软件协同与资源共享、环境感知与自主适应, 以及持续演进与质量保障 3 个方面. 当前, 随着软件的可成长性成为研究的关注点, 该系列的研究也开始探索可成长网构软件的理论、方法和技术, 进入了新的阶段.

我们考虑可成长网构软件应该具有 4 个特征, 将其总结为 4 个 “明确”, 即明确考虑业务逻辑和适应逻辑、明确建模环境感知和异常处理、明确支持版本更替和无缝演化, 以及明确管理感知效率和演化质量. 由此, 可成长网构软件将显式关注并明确实现软件的适应变化、版本演化和开发者支持, 呼应我们先前表述的软件可成长性的内涵和范畴.

3.2 实现可成长网构软件的可能途径

由以上分析可知, 可成长网构软件是一种新的软件范型, 它需要在设计机制上考虑适应变化、版本演化和开发者支持. 为了实现具备这样特征的可成长网构软件, 我们考虑 3 条可能的途径, 它们分别需要回答如何为可成长软件构建提供体系支撑, 如何使已有软件具备可成长能力, 以及如何使上述过程智能化和自动化这 3 个问题. 实际上, 对这 3 个问题的回答将解读可成长软件应该是什么样的形态 (即 “为何形态”), 如何让软件从不可成长变成可成长或增强其成长能力 (即 “如何转变”), 以及实现软件成长的自动化 (即 “智能支持”). 通俗地说, 这 3 个问题大致等价于: 天生的可成长软件长什么样, 如何改造传统软件变成这样, 以及怎样自动做到这点.

为何形态. 要回答如何为可成长软件构建提供体系支撑, 我们需要研究可成长软件的范型、机理和方法学. 具体而言, 这条途径从架构出发, 研究可成长软件之架构模型、运行机理、构造方法和质量保障, 使软件具备自主适应和持续演化能力, 实现其感知、适应和学习等智能化手段.

如何转变. 要回答如何使已有软件具备可成长能力, 我们需要研究软件可成长性的挖掘和增强技术. 具体而言, 这条途径从资源出发, 研究应用资源使用和资源依赖关系的挖掘和理解, 以及增强软件生存能力的自适应配置调整.

智能支持. 要回答如何使上述过程智能化和自动化, 我们需要研究数据驱动的软件构造和成长技术. 具体而言, 这条途径从数据出发, 实现数据驱动的软件构造和演化规律及模式获取, 基于软件演化历史数据, 在新的演化任务中实现自动化.

以上 3 条途径在逻辑上是连贯的, 同时又是 3 种实现方式, 分别考虑了如何重新设计可成长软件、如何改造成为可成长软件, 以及如何革命性地直接合成新软件. 它们分别由南京大学、中国人民解放军国防科技大学和北京大学团队主要研究, 而我们南京大学团队主要关注第 1 条途径, 以阐明可成长

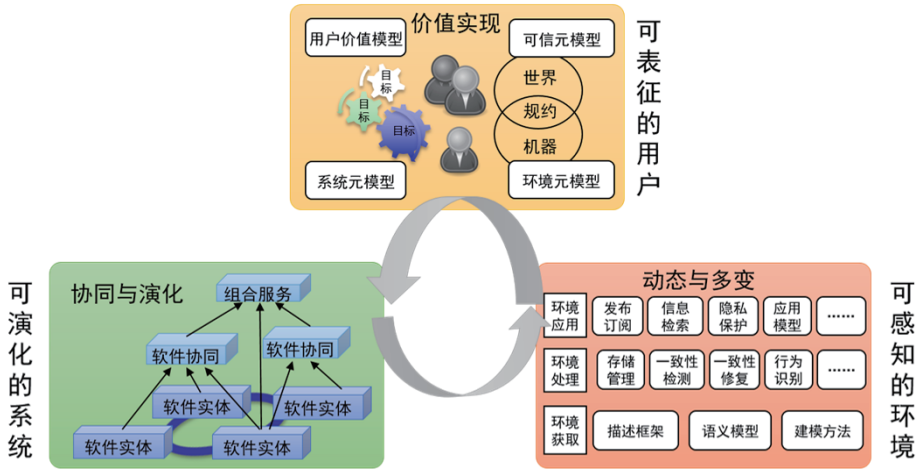

图 1 (网络版彩图) 可成长网构软件架构模型图

Figure 1 (Color online) Architectural structure of growing software

软件的范型机理 (即它到底如何运作), 以及分析其构建思路和技术挑战. 接下来我们从可成长网构软件的角度深入分析这条实现途径, 这一途径是可成长网构软件理论方法的核心, 也是其他实现途径的目标.

3.3 可成长网构软件的范型机理

如前所述, 可成长网构软件是新的软件范型, 因此我们从历史出发, 回顾和分析近年来软件范型的演化过程, 为可成长网构软件摸清脉络和总结特征.

20 世纪七八十年代, 软件范型研究者多关注于面向软件功能结构的结构化范型, 有代表性的、图灵奖级别的大师包括荷兰的 Edsger Wybe Dijkstra, 美国的 Donald Ervin Knuth, 英国的 Charles Antony Richard Hoare, 以及瑞士的 Niklaus Emil Wirth. 21 世纪初, 又涌现出一批关注面向软件问题结构的对象化范型的大师, 包括挪威的 Ole-Johan Dahl 和 Kristen Nygaard, 以及美国的 Alan Curtis Kay 和 Barbara Liskov. 我们看到, 范型的发展趋势与软件的使命有关, 其逐渐从以软件自身为中心转变为以问题解决为中心. 近年来, 软件的复杂性进一步提升, “人月神话” 的问题仍未突破, 随着 “人 – 机 – 物” 三元融合的应用模式加速发展, 其环境、资源和需求不断变迁, 软件的持续演化和长期生存逐步成为新的需求. 对此, 中国学者也从十多年前, 开始研究面向开放协同与适应演化的网构化范型, 以支持在可感知环境反馈下的系统演化. 比如国内中国人民解放军国防科技大学王怀民教授团队 [14, 15] 特别关注了局部自治系统融合成为整体软件系统的构建和维护问题, 分析了传统 “还原论” 软件开发方法的不足, 在构建方面提出了以自主化软件单元进行模块更新和连接调整的成长性构造思想, 在维护方面提出了基于统计确定性和逻辑确定性的规律寻找方式, 以及基于 “监控 – 分析 – 决策 – 调整” 的演化实现机制, 成为本文工作的基础之一.

顺此历史发展趋势和软件使命变迁, 我们总结可成长网构软件的范型机理应具备 3 项基本特征, 即三元融合的架构模型, 自主适应的运行机理, 以及持续演化的生命周期.

三元融合的可成长软件的架构模型 (图 1). 从架构模型上看, 可成长软件需要考虑用户 (人)、环境 (物) 和系统 (机) 三者的互动, 以及其三元融合下应用的抽象. 因此, 可成长范型在架构模型上包括 3 部分: 其一是 “可表征的用户”, 代表应用的价值体现; 其二是 “可感知的系统”, 代表对动态、多

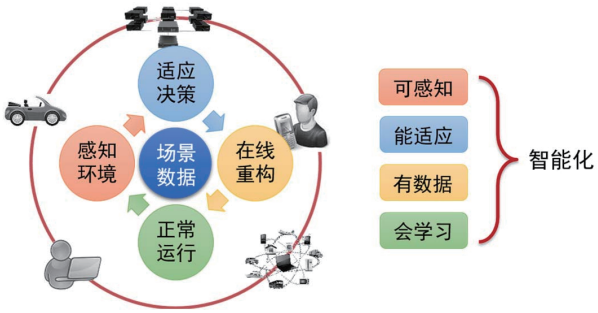

图 2 (网络版彩图) 可成长网构软件运行机理图

Figure 2 (Color online) Runtime mechanism of growing software

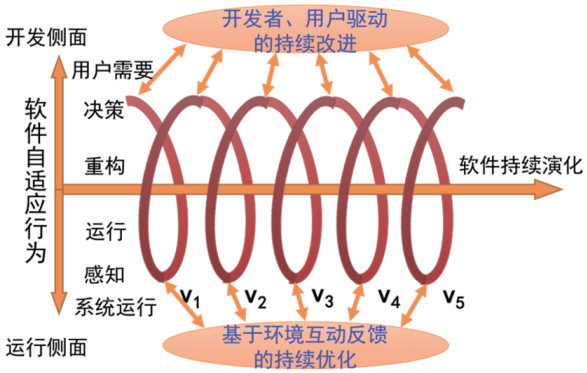

图 3 (网络版彩图) 可成长网构软件生命周期图

Figure 3 (Color online) Lifecycle of growing software

变环境的建模和理解; 其三是 “可演化的系统”, 代表软件的协同和演化能力. 这 3 部分联合一起, 在用户方面, 实现用户应用价值导向的运行时需求模型; 在环境方面, 实现基于先验元级模型和规约的环境感知; 在系统方面, 实现具有在线适应和演化能力的系统. 由此, 这 3 部分共同形成 “人 – 机 – 物” 三元融合的可成长软件的架构模型.

自主适应的可成长软件的运行机理 (图 2). 从运行机理上看, 可成长软件的运行态表现为一个迭代式的闭环适应圈, 包括 “感知环境 – 适应决策 – 在线重构 – 正常运行” 这 4 个基本环节. 在感知方面, 可成长软件主动感知环境、资源和用户需求的变化, 以知晓 “何时演化”; 在决策方面, 可成长软件进行应用目标和场景数据导向的适应决策, 以确定 “演化什么”; 在实施方面, 可成长软件实现高效、安全的运行时系统调整和更新, 以回答 “如何演化”. 由此, 这 3 方面共同形成自主适应的可成长软件的运行机理.

持续演化的可成长软件的生命周期 (图 3). 从生命周期上看, 可成长软件逐步推进一个双维度弹性的持续演化过程, 一方面基于环境的反馈进行自主适应优化 (在弹性限度之内), 另一方面基于开发者和使用者的驱动进行过程演进优化 (在弹性限度之外). 从生态方面看, 可成长软件具备一个开放的网络软件开发和运行生态环境; 从动力方面看, 可成长软件做到开发者、使用者反馈和主动感知并举; 从演进方面看, 可成长软件无缝、透明地对软件持续改进和优化. 由此, 这 3 方面共同形成持续演化的可成长软件的生命周期.

需要说明的是, 在以上架构模型中, “人” 一方面指模型中 “可表征的用户”, 因为可表征的用户代表应用的价值体现, 所以人作为应用需求的直接提出和要求变迁者, 是架构模型中非常重要的一环, 因此建模为需要明确体现其相关应用价值的可表征用户. 另一方面, “人” 也可以隐式融入到架构模型的 “可感知的系统” 和 “可演化的系统” 中, 对于前者, 比如人可提供知识、行为识别和隐私数据等, 成为环境上下文的一部分, 对于后者, 比如人可参与到软件动态更新安全点的选择和转换函数的合成与审核中, 成为演化环节的一部分. 这两方面综合起来, “人” 对于未来的软件系统, 既可以是提出要求的用户, 也可以是解决问题的关键, 因此他们是 “人 – 机 – 物” 融合应用整体系统不可或缺的一部分. 此外, 在以上运行机理和生命周期部分, 我们介绍的适应和演化是两种重要的成长方式, 前者是指基于环境的反馈进行自主适应的优化, 这是在运行态, 主要指利用软件自身已实现的功能和做法应对环境和资源的变迁, 在弹性限度之内可以无需修改软件设计和代码, 并依旧保持原先的服务质量和内容; 后者是指基于开发者和使用者的驱动进行过程演进的优化, 这与开发态和运行态都有关, 开发态包括支持在弹性限度之外的新功能设计和故障修复等行为, 运行态包括支持新版本软件进行动态更替等行为. 这两方面综合起来, 我们说, 适应和演化都是实现软件成长的重要途径.

以上关于架构模型、运行机理和生命周期 3 个基本特征分别分析了可成长软件的静态结构、运行机制和演进历程, 回答 “可成长软件是什么” 这一问题, 并给出理想的可成长软件的设计蓝图. 对照此蓝图, 接下来的重要问题是, 如何去构建这样的可成长软件, 其中有何技术挑战. 下面我们探讨并回答 “可成长软件怎么做” 这一问题.

3.4 可成长网构软件的构建思路和技术挑战

我们以元级化和定义化的思路来构建可成长软件, 这着眼于复杂的环境开放性, 其中包括应用场景的多样性、环境和需求的多变性, 以及系统内外开发者和用户的不确定性. 我们以元级化提供可成长能力, 因为传统软件属于 “目标级”, 仅实现针对给定需求和环境的功能, 而新增 “元级” 则使软件可抽象可成长能力, 以刻画感知、适应和演化行为. 元级化有个重要的哲学问题, 即认清 “我是谁”. 另一方面, 我们以定义化实现软件的成长, 以 “软件定义软件” 的方式, 克服 “软件不软” 的困难, 支持软件持续的适应和演化, 以应对环境和需求变化. 定义化也有个重要的哲学问题, 即认清 “我从哪里来, 要到哪里去”.

相反, 软件若直接实现应用功能, 则可能表现脆弱, 无论是应用需求有变迁, 或是环境和资源不再满足其功能实现的前提, 都可能造成软件难以保持原先的服务质量和内容, 从而不得不回归维护或重新构建. 而 “元级化” 概念的提出, 要求软件在设计过程中明确考虑环境、资源和需求的变迁问题, 以软件定义的方式设计实现软件在运行时的自主适应和开发时的持续演进, 也即要抽象软件的成长能力. 另外, 以上关于 “软件不软” 是形象的说法, 意思是软件若直接实现应用功能则表现 “脆而硬”, 容易被环境、资源和需求的持续变迁所打倒, 而 “软件能软” 的说法则是相反, 是期望其能在应对环境、资源和需求变迁方面游刃有余, 表现出一定程度的韧性, 这也是可成长软件的目标. 根据以上元级化和定义化的构建思路, 我们提出面向自适应和持续演化的可成长软件体系结构 (图 4). 基于此结构, 可成长软件通过基于先验元模型的环境需求建模和规约感知环境和需求变化, 通过软件定义体系结构模型支持软件系统的自适应和持续演化行为. 从图中可以看出, 这一结构分 “静态构成” 和 “运行机制” 上下两层, 从功能上又分 “环境和需求的感知与理解” 和 “软件自适应与持续演进” 左右两部分. 从静态构成上, 软件对物理环境进行抽象, 获得 “先验” 的环境和需求 (即模型和规约), 然后在运行机制上, 以环境处理中间件对物理环境进行感知, 提炼出应用上下文. 以静态规约和运行时上下文共同驱动可适应的软件系统, 以适应逻辑推动业务逻辑的优化, 实现系统的适应和演化行为. 这样的规划在整体上体

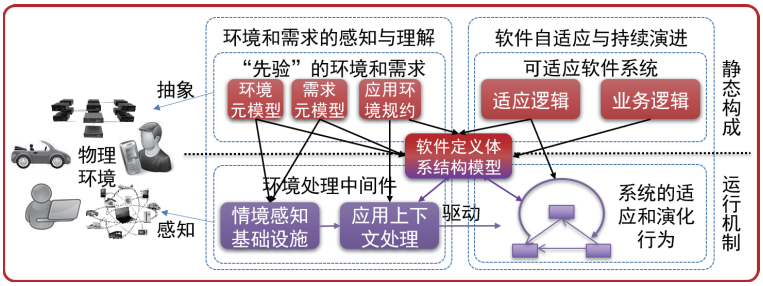

图 4　(网络版彩图) 面向自适应和持续演化的可成长软件体系结构

Figure 4　(Color online) Architecture of growing software with self-adaptation and continuous-evolution support

现了软件定义的系统结构模型, 其中上层 (红色) 是元级部分, 下层 (紫色) 是新的目标级部分.

有了前述的可成长软件设计蓝图和构建思路, 我们回答 “可成长软件怎么做” 这一问题, 还需解决众多的技术挑战. 比如说, 在运行时刻, 如何做到环境上下文的高效处理, 在演化时刻, 如何做到软件版本的无缝更替, 以及在整个过程中, 如何评估和保障环境上下文处理和软件适应演化的质量等. 针对这些挑战, 我们在第 4 节中探讨支撑软件可成长性的关键技术, 并分析和总结当前的研究进展.

4　支撑软件可成长性的关键技术

本节讨论支撑软件可成长性的关键技术, 包括赋能关键技术和外围关键支撑, 其涉及软件开放环境感知与自适应、无缝演化和过程演进, 以及演化质量评估方法和保障机制等重要方面.

4.1　软件可成长性关键技术及其支撑角色

支撑软件可成长性的关键技术主要覆盖可成长网构软件的几个重要方面, 包括开放环境感知与自适应方法和技术、软件无缝演化和过程演进支撑技术, 以及软件演化质量评估方法和保障机制. 其中, 开放环境感知与自适应方法和技术负责准确、高效地理解环境和资源的变化, 软件无缝演化和过程演进支撑技术负责软件在线更新并挖掘用户需求的变化, 软件演化质量评估方法和保障机制负责评估和保障环境质量和软件一致性演化.

开放环境感知与自适应方法和技术. 这是支撑可成长软件内化自主适应能力的核心技术, 需要做到: 高效的环境上下文一致性处理, 以提升检测和修复环境上下文一致性错误的处理性能; 资源使用的在线监控和优化, 以从资源配置和调度角度为软件运行提供有效资源支撑和性能优化; 以及多维多粒度的资源调度优化, 以为异构资源进行抽象并建立优先级资源调度. 这方面的几项技术协作实现准确、高效理解环境和资源变化的目标.

软件无缝演化和过程演进支撑技术. 这是支撑可成长软件内化过程演进能力的核心技术, 需要做到: 高一致、低干扰的软件在线更新和版本热部署, 以提供一致性保障并最小化对服务干扰; 软件运行环境的可配置定义, 以实现软件在线演化中的资源保障; 以及面向开发生态的运维推荐和用户需求分析, 以支撑智能化开发成员推荐和用户隐式需求分析. 这方面的几项技术协作实现软件在线更新并挖掘用户需求变化的目标.

软件演化质量评估方法和保障机制. 这是支撑可成长软件持续演化服务质量评估和保障的核心技术, 需要做到: 环境上下文一致性处理效率和效果的科学评估和保障, 如以处理性能、漏报数量、误报数量和修复效果评估质量, 以增量式检测、并行式检测和跨越式调度保障质量等; 软件在线更新安全

性和效率的科学评估和保障, 如以服务一致性, 对服务干扰程度、停顿时间和性能损失评估质量, 以构件级一致性维护、代码级更新错误恢复和系统级更新测试保障质量; 以及基于模型的持续演化策略构造及验证, 如以层次化自适应和在线学习规划解决不确定性和不可预见性问题. 这方面的几项技术协作实现评估和保障环境质量和软件一致性演化的目标.

我们把支撑软件可成长性的关键技术分为赋能关键技术和外围关键支撑两部分, 前者直接负责软件开放环境感知与自适应、无缝演化和过程演进, 以及演化质量评估方法和保障机制 3 个重要方面, 后者从静态分析、动态检测和安全分析 3 方面辅助支撑软件的可成长性. 接下来我们分析和总结这些关键技术的当前研究进展.

4.2 赋能关键技术的研究进展

针对软件开放环境感知与自适应、无缝演化和过程演进, 以及演化质量评估方法和保障机制 3 个重要方面的赋能关键技术, 我们先分析其解决思路, 这大致可分为可成长软件内化自主适应能力与质量保障, 以及可成长软件内化过程演进能力与质量保障这两部分.

可成长软件内化自主适应能力与质量保障. 软件要准确、高效地理解环境和资源的变化, 就必须应对其从环境和资源处获得不精确、不完整, 甚至相互冲突上下文数据的技术挑战, 这要求有高效的环境上下文一致性处理技术. “一致性处理” 意味 “看得透”, 即要确保上下文数据的质量; “高效” 意味 “做得快”, 要能在软件运行时高性能地检测和修复上下文数据的错误. 我们的目标是实现环境上下文处理效率的数量级提升, 解决思路是细粒度多维度优化. 具体而言, 对每次约束检测的调度, 如果上下文数据前后关联, 我们可采用强耦合增量式约束检测, 达成 “单步更高效”, 如果数据前后无关, 我们可采用弱耦合并行式约束检测, 达成 “多步能并行”, 以双管齐下, 实现单次调度加速的效果; 对多次约束检测的调度, 我们可识别其中的无效调度 (即不检测也无漏报错误), 采用跨跃式检测调度, 达成 “跨越批处理”, 以实现多次调度加速的效果. 两方面综合起来, 我们在约束检测调度内和调度间分别进行细粒度和多维度的优化, 可期望实现环境上下文处理效率的数量级提升.

可成长软件内化过程演进能力与质量保障. 软件要支持在线更新并挖掘用户需求的变化, 就必须解决 “怎么变” 的技术问题和 “变什么” 的需求问题. 对前者, 要求有高一致、低干扰的软件在线更新和版本热部署技术, 对后者, 要求有面向开发生态的运维推荐和用户需求分析技术. 在软件在线更新方面 (即 “怎么变”), 我们的目标是实现更新过程的服务一致性, 并降低对服务的干扰程度、停顿时间和性能损失, 解决思路是在线式多层面更新. 具体而言, 在架构层, 我们可利用依赖分析支持过程的动态迁移; 在构件层, 我们可基于动态依赖降低对服务的干扰; 在代码层, 我们可使用延时更新缩短服务的中断时间. 而在用户需求挖掘方面 (即 “变什么”), 我们既可从社交活动中分析用户的需求, 即通过软件使用行为和评价信息分析用户潜在需求并辅助以用户具体评论文本构建用户显式需求, 也可从日常行为中推断用户的需求, 即在感知层收集用户可观测行为、心理和生理数据, 在融合层分析多源异构数据, 解决其中冗余、噪声和数据缺失等问题, 在理解层建立用户行为、思维和社会关系模型. 此外, 既与 “怎么变” 又与 “变什么” 密切相关的一个技术问题是如何面向开发生态进行正确、合适的运维推荐 (即由谁来解决如何开发和维护问题), 我们的解决思路是利用开发者能力建模和项目成员推荐, 即通过开发者的项目经历建模其个体能力和协作方式, 之后在项目发生人员变动时可推荐最适合此项目团队的成员.

以下我们结合这两部分的解决思路, 分析和总结目前在开放环境感知与自适应方法和技术、软件无缝演化和过程演进支撑技术, 以及软件演化质量评估方法和保障机制这 3 方面赋能关键技术上的进展. 限于篇幅, 我们在每个技术点上仅列举一至两项代表性工作.

4.2.1 开放环境感知与自适应方法和技术

这方面研究要解决高质高效的环境上下文一致性处理问题.

可成长软件需要能够根据所感知的环境上下文调整自身行为, 以实现智能的软件服务, 但环境上下文中往往包含各种感知噪音和测量误差, 容易导致错误的行为调整, 甚至造成软件崩溃. 当前的方法是通过约束检测发现环境上下文一致性错误并及时修复, 但已有技术采用即时调度策略, 虽然能保证正确性但是效率低下, 也有尝试批量检测策略, 虽然能提高效率但是丢失率居高不下. 对此, 我们提出了基于自适应分组的批量检测策略 GEAS [16], 既保证正确性, 又兼容已有约束检测技术并大幅提升检测效率. 与已有工作相比, GEAS 可提升约束检测效率 1.4~6.5 倍, 并保证零丢失率, 而已有工作则会导致 39%~65% 的丢失率. 在此基础上, 基于组内匹配对消优化, 我们进一步提出了跨越批处理式增强型检测策略 GEAS-opt [17], 在零丢失率前提下, 提升约束检测效率 6.7~28.6 倍. 基于该技术, 我们实现了原型测试系统, 使用 588 万城市车辆 GPS 数据开展一致性约束检测实验, 结果表明该系统在离线检测配置下可提升检测效率 48~148 倍, 在在线检测配置下可提升检测效率 18~42 倍, 并保证了零漏报率和零误报率, 有效支撑了可成长软件高质高效感知开放环境的能力.

4.2.2 软件无缝演化和过程演进支撑技术

这方面研究要解决高一致低干扰的软件动态更新, 非确定环境下的自适应系统验证, 以及面向开发生态的运维推荐和需求分析等问题.

可成长软件的无缝演化依赖于动态更新技术, 它可以在不终止正在运行软件系统的前提下将其更新到新版本. 在动态更新的过程中, 被修改系统组件的运行时状态需被合理地转换到对应的新状态, 以确保被修改组件仍可正确与其他组件交互. 然而实现运行时状态转换面临诸多困难, 主要是由于不同版本组件的底层实现差异所导致. 对此, 我们提出了 AOTES 技术 [18], 针对 Java 程序实现动态更新中的自动对象状态转换. 为了消除不同版本实现层的差异, AOTES 将旧版本中对象状态抽象成方法调用历史, 然后将该调用历史中的旧版本方法替换成对应的新版本方法, 并重新执行新版本方法调用序列创建对应的新版本对象状态. AOTES 不需要对被更新程序做任何代码插桩, 因此在软件系统常态执行过程中没引入额外负载. 我们收集了若干开源软件开发库和服务器程序, 包括 Apache Commons Collections, Tomcat, FTP 以及 SSHD. 在 61 个实验对象转换中, AOTES 可成功完成 51 个对象转换, 而已有技术 (如缺省转换和 TOS 转换 [19]) 只能处理 6~11 个对象转换. 在此基础上, 我们分别在代码层、构件层和应用层部署软件动态更新技术. 针对重要的技术指标, 在干扰减少方面, 我们初步实现面向微服务的软件无缝演化, 较传统安全的 Quiescence 方法 [20], AOTES 在版本更新时可减少 23.9% 的干扰, 在停顿控制方面, 我们初步实现面向代码级更新的 Javelus 8, 在 42 个真实 Tomcat 更新上的测试实验中, 系统中断时间不超过 70.5 ms, 有效支撑了可成长软件的跨版本无缝演化能力.

此外, 可成长软件的自主适应需要应对开放环境的非确定性, 因此非确定环境下的自适应系统验证是一个重要问题. 已有技术通过反例构造验证和发现自适应系统中的缺陷问题, 但其所依赖的非确定环境模型往往不够准确而需要在真实环境中对构造的反例进一步确认. 困难是这样构造的反例由于发生概率很低而导致确认的效率极低, 严重影响系统验证的效果. 对此, 我们提出了 VERSA 技术 [21], 以原有反例为基础, 进一步构造出新的路径等价反例, 这些新构造的反例能确保触发等价的原缺陷问题, 但拥有基于当前非确定环境模型的最大发生概率. 实验表明, VERSA 比已有技术可显著提升构造反例的发生概率 (发生概率提升一个数量级, 触发时间降低一个数量级), 有效推进了自适应系统的缺陷验证水平. 进一步的, 此类方法其所依赖的非确定环境模型由于感知误差和数据样本问题往往不够

准确, 可能影响系统验证的正确性. 对此, 我们改进了 VERSA 技术 [22], 基于构造反例的预估概率和实际发生概率的偏差, 识别非确定模型中的不准确性, 并以最小化偏差为导向系统化地校准非确定模型, 并将其归结为一个基于搜索的软件工程问题予以解决. 实验表明, VERSA 可大幅提升非确定环境模型的准确度 (绝对值提升 30.9%, 相对值提升 120.3%), 进一步推进了自适应系统的缺陷验证水平, 支撑了可成长软件的过程演进质量.

4.2.3 软件演化质量评估方法和保障机制

这方面研究要解决软件适应与演化质量保障方法和工具集等问题. 前述的环境感知和软件演化技术也蕴含针对可成长软件适应与演化质量的评估方法和保障机制, 覆盖了对环境上下文一致性错误的检测效率, 动态软件更新的一致性和停顿时间等方面的考虑. 与此同时, 可成长软件的适应与演化也需要关注其他侧面的质量保障工具集, 比如考虑如何触发并发缺陷、诊断能耗异常、检测恶意打包、合成正则攻击、检验场景适配、发现服务泄露, 以及测试深度学习等各个方面, 这些侧面也都蕴含重要的质量保障技术. 下面我们介绍其中两项工作.

可成长软件的思想直接适用于服务类软件系统, 因为可以支撑以提供长久有效的服务. 然而, 现代服务类软件通常采用正则表达式处理用户输入的字符串信息, 而设计有误的正则表达式可引发指数级的匹配复杂度, 进而导致拒绝服务式的攻击. 那么, 如何判断服务类软件的一个新版本是否修复了这样的问题? 我们提出了 ReScue 技术 [23] 进行分析, 它解决了输入的长前缀和攻击串之间特征不匹配的问题, 以及检测时间有限和串演化缓慢之间的矛盾, 利用三阶段灰盒分析自动产生针对给定正则表达式的攻击字符串. 实验表明, ReScue 可提升攻击字符串生成率 49%, 并实际检测到真实服务类软件中前所未知的多个正则表达式攻击缺陷, 有效提升了服务类软件的安全性. 这项工作也获得了 ASE 2018 ACM SIGSOFT 杰出论文奖.

此外, 采用机器学习模块参与可成长软件决策行为的趋势在逐步增强, 而机器学习模块往往基于深度神经网络 DNN 搭建, 如何保障 DNN 的可靠性? 测试这类软件通常借用测试常规软件中的结构化覆盖率标准, 以方便评估测试的充分性. 然而, 我们发现这类标准可能具有误导性, 因为 DNN 软件和常规软件有着本质区别. 我们的初步调研 [24] 发现: (1) 结构化覆盖率标准对发现 DNN 软件的恶意输入可能太粗粒度了 (容易满足其标准却不能实质性提升检错能力), 而对于发现误判的正常输入则可能太细粒度了 (很难满足其标准, 即很难发现真实缺陷); (2) 已有的 DNN 软件测试显得覆盖率高, 很可能是因为其基于面向恶意输入的搜索, 而要达成找到很多恶意输入的效果 (显得检错能力高), 实际是很容易做到的 (无关真实的覆盖率). 基于该调研, 我们把 DNN 软件输入界定为预期输入和非预期输入两类, 而需要着重保障可靠性的是预期输入. 对此, 我们提出了基于容错思想的 DISSECTOR 技术 [25], 自动识别并隔离超出原模型处理能力的未预期输入, 以保证原应用仍能保持所期望的可靠性. 实验表明, DISSECTOR 能对 DNN 模型自动适应, 并对超大数据应用场景非常高效 (高至 0.99 的有效性, 可提升模型准确度 16%∼20%, 以及仅有 3∼6 ms 的运行时间), 有效增强了基于机器学习决策的可成长软件的可靠性.

4.3 外围关键支撑的研究进展

软件可成长性也需外围支撑的辅助, 下面我们介绍关于静态分析和模型生成、动态检测和智能决策, 以及应用探索和安全验证 3 方面的需求, 限于篇幅, 各列举一至两项代表性工作.

4.3.1 软件静态分析和模型生成

这方面研究要解决软件制品构造和演化途径查询等问题.

可成长软件视原有的软件设计和代码为财富, 将其转化为 “模型 – 软件制品库”, 以有效支撑未来软件的构建和演化. 为此, 需要解决如何基于现有的互联网软件资产构造出可被理解和复用的软件制品库, 以及为基于此设想的软件构造和演化提供相应的机制和技术. 对前者, 我们以开源软件为切入点, 通过逆向工程, 如软件动态和静态分析等手段, 理解每个方法并抽取不同抽象层次的语义信息, 然后基于工业建模标准 UML 家族规范和扩展机制, 构造出模型制品库. 同时, 我们保持模型制品和软件资产 (包括源代码、配置文件、文档、开发日志、运行时截屏, 以及执行日志等) 之间的对应关系, 以形成可被多维度理解和复用的 “模型 – 软件制品库”. 对后者, 我们基于此制品库研究面向代码层面的查询、综合和复用等技术, 以及系统层面测试、安全检测和不确定性行为理解等技术. 在此过程中, 我们形成了一系列程序分析与测试、模型扩展、隐私泄露检测, 以及代码查询与合成等方法和工具, 对软件构建和演化的支撑效果显著, 如有效支撑了面向高精度数值型程序的演化 [26].

4.3.2 软件动态检测和智能决策

这方面研究要解决任务协调部署和资源动态分配等问题, 这些问题与当前蓬勃发展的大数据和云计算技术背景密切相关. 可成长软件的适应决策越来越依赖于大数据分析系统, 此类系统的资源利用率和作业性能变得至关重要. 现有的作业调度方式假设了集群知道完全的作业先验知识, 而当有新类型应用到来或集群产生资源异常情况时, 完全先验知识的假设就会失效. 对此, 我们设计了基于部分先验知识的作业管理器, 能够实现乘性的实时资源需求调整, 并在集群管理器中采用公平的调度方式, 相较于系统原有的作业调度方式有约 30% 的性能提升 [27]. 此外, 可成长软件的应用模式正出现跨域云计算的趋势, 地域分布的异构边缘集群与数据中心构成了 “云 – 边缘系统”. 一方面, 传统按定额分配资源的方式无法满足上层应用的各式需求, 另一方面, 云数据中心和边缘集群的资源差异明显, 传统资源部署和任务分配机制容易产生资源使用瓶颈. 对此, 需要在资源管理软件中进行智能决策, 结合上层应用的资源需求, 协同底层数据和资源分布进行任务协同部署. 我们提出了智能数据任务协同部署机制和动态资源分配策略 [28], 可提升集群利用率, 摆脱性能瓶颈, 最终提升用户体验. 目前, 我们初步完成了基于 Spark 和 Hadoop Distributed File System 的开源源码修改, 形成了云服务器跨节点验证测试实验床, 可降低大数据处理时延, 引导出更好的数据分布. 这些努力有助于支撑大数据和云计算技术背景下的软件任务协调部署和资源动态分配, 辅助可成长软件基于运行时的动态检测结果做出智能决策.

4.3.3 软件应用探索和安全验证

这方面研究探索以轨交列控系统的节能适应和安全演化做应用验证.

我们构建了轨道交通列控系统运营环境监控、能耗状态感知、评估和预警平台暨演化例证平台, 运用可成长网构软件理论方法优化了系统架构, 并构建了列车运行控制仿真试验系统. 该系统平台以南京地铁机场线、三号线和宁天线节能适应项目开展实例验证分析和优化仿真测试, 通过调节列车在站停靠时间和发车间隔等方式以优化列车运行图, 并调节列车运行速度曲线和追踪距离以优化列车运行等级, 可综合实现节能 5%∼8% 的目标 (目前每条线路电耗标准约 4000 万度) [29].

此外, 轨交列控系统的区域控制器代码由于规模问题无法被现有方法和工具有效验证, 在安全演化方面隐患很大. 对此, 我们提出了两种基于问题框架的验证系统预处理方法. 第 1 种, 采用子安全属性投影, 按行业标准 (如 IEEE 1474.1 和 IEEE 1474.3) 对安全属性进行分解, 获得子安全属性集, 并以

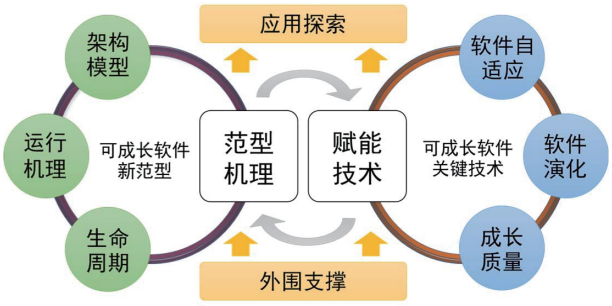

图 5 (网络版彩图) 可成长网构软件理论方法和技术的内在关联

Figure 5 (Color online) Terminologies and relationships of theories and techniques of growing software

此作为投影维度, 进行分解和组合验证. 第 2 种, 采用约束投影, 根据领域知识获得验证系统变量的约束, 并通过分析这些约束得到一个变量集合作为投影维度以对系统进行投影. 对小规模系统进行验证时, 耗时可缩短 60%; 当应用于大规模状态空间的实际案例 (如卡斯柯信号有限公司上海地铁 17 号线区域控制器移动授权计算模块) 时, 我们方法可减少高达 80% 的验证状态空间, 使得原本无法分析的系统得到了成功验证 [30], 有效保障了轨交列控系统核心模块在安全演化方面的可靠性.

5 可成长软件理论方法和技术的集成应用

前面介绍和讨论的可成长网构软件理论方法和关键技术, 涉及软件构建和维护的方方面面, 力图从各个角度支撑内化自主适应和持续演进能力的软件成长. 下面我们进一步分析这些理论方法和技术的内在关联, 给出集成应用的建议, 并总结这一理论方法和技术体系的创新点.

首先, 图 5 展示了可成长网构软件理论方法和技术的内在关联. 如图所示, 先前介绍和讨论的各项工作, 在整体上基于网构软件范型, 从架构模型、运行机理和生命周期这 3 个方面进行了拓展, 形成了支持软件成长的新范型, 并提供了软件自适应、持续演化和成长质量保障等必须的关键支撑技术. 从一方面看, 软件范型机理研究覆盖了架构模型、运行机理和生命周期, 从整体上阐明了软件可成长性的内涵, 分析了可成长范型机理的基本特征, 回答了 “可成长软件是什么” 这一问题. 从另一方面看, 赋能技术研究包括了软件自适应、持续演化和成长质量保障, 加上外围支撑和应用探索, 共同支持了可成长软件的有效构建和后期维护, 回答了 “可成长软件怎么做” 这一问题.

其次, 图 6 进一步展示了如何集成应用这些理论方法和技术. 一般而言, 应用软件在设计上可以通过环境资源监控与优化识别环境和资源的变化, 并通过用户需求分析与挖掘识别用户需求的变化, 结合这两方面, 共同应对环境、资源和需求的持续变迁 (即明确应对变化). 在此基础上, 应用软件可以执行一个 “理解 – 规划 – 演化” 的过程来持续推进软件的成长 (即明确推进演化). 具体的建议包括: 以环境上下文处理来高质高效地理解环境、资源和需求的变化 (变化的引入), 以多粒度多方位适应决策来规划弹性限度之内的软件自主适应 (适应的推进), 以及以软件在线更新来实现弹性限度之外的无缝版本演化 (过程的演进). 此外, 在软件的成长过程中, 可部署前面介绍的软件适应与演化质量保障工具集来持续监控和维护软件成长的平稳性和安全性. 当然, 上述集成应用是在方法论层面的建议, 具体操作和实践已部分体现在本项目多个团队的工作中. 比如, 中国人民解放军国防科技大学团队在软件演化过程中, 以智能日志增强的方式挖掘软件中代码的意图 (需求变化提取), 从而提升软件演化的效率 (自动新增代码); 中国科学院软件研究所团队在云端服务软件的更新过程中, 根据业务场景变

图 6　(网络版彩图) 可成长网构软件理论方法和技术的集成应用

Figure 6　(Color online) Integrated applications of theories and techniques of growing software

化 (环境变化提取), 以订阅和容错机制提升软件配置动态更新的效率和可靠性 (软件配置适应); 中国电子科技集团公司第二十八研究所团队在面向高动态战场环境的指挥控制系统中, 以 “环境认知 – 决策执行” 双环自主适变机制 (从感知到适应和演化), 实现资源受损环境下的模块适应和等级转进需求下的版本升级. 更详细的解读可参考这些团队的具体工作.

再次, 我们考虑可成长网构软件理论方法和技术应用的目标场景, 从技术工具平台的支撑方面看, 它们比较适用于两类软件的成长, 即云端服务软件和嵌入端/移动端软件. 这两类软件在 “人 – 机 – 物” 三元融合大趋势下, 对自适应和持续演化的需求较为突出 (如要求持续服务和开放环境等), 而可成长网构软件理论方法和技术对它们的支撑也比较明确. 比如说, 为云端服务软件自适应和持续演化服务的支撑平台与工具链, 包括来自可成长软件实现途径一的构件级在线更新支撑工具和程序级在线更新支撑工具 (南京大学团队主导), 以及来自系统平台研究的应用意图分析工具、微服务演化框架、平台资源调度框架、应用容错管理框架, 以及软件定义的应用容器引擎 (中国科学院软件研究所团队主导); 而为嵌入端/移动端软件自适应和持续演化服务的工具链, 则包括来自可成长软件实现途径二的资源使用分析工具与资源依赖和冲突分析工具 (中国人民解放军国防科技大学团队主导), 以及来自实现途径三的用户偏好识别工具与应用流行度预测工具 (北京大学团队主导). 除此以外, 来自可成长软件实现途径一的环境上下文处理工具 (南京大学团队主导), 以及来自实现途径三的程序缺陷自动修复工具和程序代码自动生成工具 (北京大学团队主导), 可作为通用技术工具, 同时为云端服务软件和嵌入端/移动端软件两方面提供基础功能和服务.

最后, 基于上述分析的讨论, 我们总结可成长网构软件理论方法和技术的创新点. 这是一种基于网构软件范型抽象的可成长软件新形态, 即以网构化的软件抽象, 克服传统范型及其方法技术体系之环境依赖固定、系统架构封闭、开发演化割裂等不足, 并通过内化自适应与持续演化支持, 实现具有长期生存不断成长能力的软件系统新形态. 我们期望这套理论方法和技术, 可促进如下几方面的重要转变: 在软件的角色界定方面, 从 “整个应用的信息处理环节” 转变为 “面向 ‘人 – 机 – 物’ 三元融合场景下应用价值观的主要载体”; 在软件的个体形态方面, 从 “以功能实现为中心, 以环境假设静态固化为特征的软件形态” 转变为 “以协同为中心, 以情境交互动态适应和持续演化为特征的软件形态”; 在软件的开发方法方面, 从 “自上而下的集中管理和封闭目标导向的过程化开发” 转变为 “开放生态下

开放共享的生长式, 数据驱动的网构化开发"; 在软件的应用运维方面, 从 "面向既定场景的应用部署和与软件开发相互割裂的运维" 转变为 "软件定义的应用场景, 相互融合的开发运维, 系统持续演化长期生存"; 在软件系统的质量保障方面, 从 "主要关注客观证据, 软件实体的正确性和质量, 当前时点的质量指标" 转变为 "关注主客观融合、应用价值导向的质量属性, 以及以感知、适应和持续演进, 长期生存能力为指标的可成长式软件质量保证". 以上这几方面的转变, 将促使软件逐步走上从基础的可成长到进阶的自成长的道路, 这是一条更具活力, 也更具挑战的软件进化之路.

6 结束语

本文介绍了软件可成长性问题的由来, 及其概念内涵和范畴, 探讨了可成长网构软件理论方法和关键技术, 并建议了集成应用方式和场景. 我们当前的努力还很初步, 只是开启了明确考虑和直接应对软件持续演化和长期生存需求的第一步. 然而, 软件对现代信息社会的重要性毋庸置疑, 特别是在如今云计算、大数据技术和 "人 – 机 – 物" 三元融合应用背景下, 软件基础设施所起的作用和面临的挑战都越来越大. 如何使软件具备自主适应和持续演进的能力, 以达到不断优化和长期生存的目标, 其解决方案将是加速社会信息化进程, 推进众多信息化和智能化行业效率的巨大引擎. 我们期待更多的行业专家和软件开发实践者一起努力, 推动可成长和自成长软件的新跨越.

致谢 感谢特约编辑、编辑部和匿名审稿人的帮助.

参考文献

1 Glass R L. Sorting out software complexity. Commun ACM, 2002, 45: 19–21
2 Marcil L, Hawthornthwaite M. Realizing DO-178C's value by using new technology: OOT, MBDV, TQC & FM. In: Proceedings of the 31st IEEE/AIAA Digital Avionics Systems Conference (DASC), Williamsburg, 2012. 1–10
3 Brooks F P. The Mythical Man-Month. Hoboken: Addison-Wesley Professional, 1975
4 Brooks F P. No silver bullet essence and accidents of software engineering. Computer, 1987, 20: 10–19
5 Beck K. Extreme Programming Explained: Embrace Change. Hoboken: Addison-Wesley Professional, 2000
6 Bass L, Weber I, Zhu L. DevOps: A Software Architect's Perspective. Hoboken: Addison-Wesley Professional, 2015
7 Gamma E. Design Patterns: Elements of Reusable Object-oriented Software. Chennai: Pearson Education India, 1995
8 Fowler M. Refactoring: Improving the Design of Existing Code. Hoboken: Addison-Wesley Professional, 2018
9 Ng T H, Cheung S C, Chan W K, et al. Work experience versus refactoring to design patterns: a controlled experiment. In: Proceedings of the 14th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE 2006), Portland, 2006. 12–22
10 Ng T H, Cheung S C, Chan W K, et al. Do maintainers utilize deployed design patterns effectively? In: Proceedings of the 29th International Conference on Software Engineering (ICSE 2007), Minneapolis, 2007. 168–177
11 Lin H M. Program automation. In: Proceedings of Workshop on Oppotunities and Challenges of Software Automation in the Era of Big Data, Yanxi Lake Conference, 2018 [林惠民. 程序自动化. 见: 雁栖湖会议之 "大数据时代软件自动化的机遇和挑战", 2018]
12 Lü J, Ma X, Huang Y, et al. Internetware: a shift of software paradigm. In: Proceedings of the 1st Asia-Pacific Symposium on Internetware (Internetware 2009), Beijing, 2009. 1–9
13 Mei H, Huang G, Xie T. Internetware: a software paradigm for internet computing. Computer, 2012, 45: 26–31
14 Wang H M, Wu W J, Mao X J, et al. Growing construction and adaptive evolution of complex software system. Sci Sin Inform, 2014, 44: 743–761 [王怀民, 吴文峻, 毛新军, 等. 复杂软件系统的成长性构造与适应性演化. 中国科学: 信息科学, 2014, 44: 743–761]
15 Wang H M, Ding B. Growing construction and adaptive evolution of complex software systems. Sci China Inf Sci, 2016, 59: 050101

16 Guo B, Wang H, Xu C, et al. GEAS: generic adaptive scheduling for high-efficiency context inconsistency detection. In: Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution (ICSME 2017), Shanghai, 2017. 137–147
17 Wang H, Xu C, Guo B, et al. Generic adaptive scheduling for efficient context inconsistency detection. IEEE Trans Softw Eng, 2019. doi: 10.1109/TSE.2019.2898976
18 Gu T, Ma X, Xu C, et al. Automating object transformations for dynamic software updating via online execution synthesis. In: Proceedings of the 32nd European Conference on Object-Oriented Programming (ECOOP 2018), Amsterdam, 2018. 1–28
19 Magill S, Hicks M, Subramanian S, et al. Automating object transformations for dynamic software updating. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA 2012), Tucson, 2012. 265–280
20 Kramer J, Magee J. The evolving philosophers problem: dynamic change management. IEEE Trans Softw Eng, 1990, 16: 1293–1306
21 Yang W, Xu C, Pan M, et al. Efficient validation of self-adaptive applications by counterexample probability maximization. J Syst Softw, 2018, 138: 82–99
22 Yang W, Xu C, Pan M, et al. Improving verification accuracy of CPS by modeling and calibrating interaction uncertainty. ACM Trans Internet Technol, 2018, 18: 1–37
23 Shen Y, Jiang Y, Xu C, et al. ReScue: crafting regular expression DoS attacks. In: Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2018), Montpellier, 2018. 225–235
24 Li Z, Ma X, Xu C, et al. Structural coverage criteria for neural networks could be misleading. In: Proceedings of the 41st ACM/IEEE International Conference on Software Engineering (ICSE 2019 NIER), Montreal, 2019. 89–92
25 Wang H, Xu J, Xu C, et al. DISSECTOR: input validation for deep learning applications by crossing-layer dissection. In: Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering (ICSE 2020), Seoul, 2020
26 Wang X, Wang H, Su Z, et al. Global optimization of numerical programs via prioritized stochastic algebraic transformations. In: Proceedings of the 41st ACM/IEEE International Conference on Software Engineering (ICSE), Montreal, 2019. 1131–1141
27 Zhang X, Qian Z, Zhang S, et al. COBRA: toward provably efficient semi-clairvoyant scheduling in data analytics systems. In: Proceedings of the IEEE Conference on Computer Communications (INFOCOM), Honolulu, 2018. 513–521
28 Wang X, Chen M, Lu S. Modeling geographically correlated failures to assess network vulnerability. IEEE Trans Commun, 2018, 66: 6317–6328
29 Song W, Jacobsen H A, Cheung S C, et al. Workflow refactoring for maximizing concurrency and block-structuredness. IEEE Trans Serv Comput, 2018. doi: 10.1109/TSC.2018.2867593
30 Yuan Z, Chen X, Liu J, et al. Simplifying the formal verification of safety requirements in zone controllers through problem frames and constraint-based projection. IEEE Trans Intell Transp Syst, 2018, 19: 3517–3528

Theories and techniques for growing software: paradigm and beyond

Chang XU[1,2*], Yi QIN[1,2], Ping YU[1,2], Chun CAO[1,2] & Jian LÜ[1,2]

1. *State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing* 210023*, China*;
2. *Department of Computer Science and Technology, Nanjing University, Nanjing* 210023*, China*
* Corresponding author. E-mail: changxu@nju.edu.cn

Abstract With advances in cloud computing and big data technologies, "human-cyber-physical" applications are providing increasingly rich information and robust functionality. This imposes new technical changes on software systems, which are required to make self-adaptation and continuous evolution to meet our increasingly higher expectations. This article explores theories and techniques for growing software to meet the challenges caused by constantly changing environments and external resources. It studies the source of the software growability problem, seeks to define software growability, and develops a paradigm for growing software. It further analyzes the challenges of supporting environmental sensing and self-adaptation, realizing seamless evolution and process optimization, and developing quality evaluation and assurance mechanisms for growing software. It also reports recent technical advances in these areas from the perspective of long-living and continuously-growing software.

Keywords growing software, paradigm, self-adaptation, continuous evolution

Chang XU received his Ph.D. degree in computer science and engineering from The Hong Kong University of Science and Technology, China. He is a full professor with State Key Laboratory for Novel Software Technology and the Department of Computer Science and Technology at Nanjing University. His research interests include big data software engineering, intelligent software testing and analysis, and adaptive and autonomous software systems.

Yi QIN received his Ph.D. degree in computer science and technology from Nanjing University, China. He is an assistant researcher with State Key Laboratory for Novel Software Technology and the Department of Computer Science and Technology at Nanjing University. His research interests include software testing and adaptive software systems.

Ping YU received her Ph.D. degree in computer science and technology from Nanjing University, China. She is an associate professor with State Key Laboratory for Novel Software Technology and the Department of Computer Science and Technology at Nanjing University. Her research interests include big data software engineering, software evolution, and cloud computing.

Chun CAO received his Ph.D. degree in computer science and technology from Nanjing University, China. He is a full professor with State Key Laboratory for Novel Software Technology and the Department of Computer Science and Technology at Nanjing University. His research interests include distributed computing platforms.