软件开发 软件定义 构造 (15)

69-73,77

计算机科学1994Vol. 21№. 6

新的软件开发方法论需求：软件定义的高层构造

应 晶 吴朝晖 何志均

(浙江大学人工智能研究所 CAD & CG 国家重点实验室 杭州310027)

摘 要 This paper puts forward a kind of novel methodolgy for software system development, from the point of view of the problem existed in the software development procedure—the gap between the requirement specification level and the program implementation level. We attempt to begin from the specification level of software development to touch the process of high-level specification construction profoundly. We propose a specification language to support multiple semantic dimensions and on this basis build a unified functional model of software system in a specific domain. Based on these, we apply transformation and refinement methods to the model and transit from the specification level to the implementation level. We expect such a process can change the current software producing procedure in nature.

关键词 Software System, Gap, Specification Level, Implementation level, High-level, Construction

一、研究背景

多年软件开发的实践，人们积累了丰富的经验，也熟悉了软件开发过程的一些实质性的关键环节。软件开发人员知道如何去协调各个阶段的开发任务，如何强调某些步骤，同时，所暴露出来的问题也对软件研究人员提出了迫切的要求。

传统的软件生命周期方法把开发过程划分为若干阶段。应该说在当时的商业性挑战面前，是一种有效的方法论。这种做法显式地强调了软件各个加工阶段的工作重点。但是这种方法实质并没有引入"流水线"开发形式。由于整个过程中大量的人工劳动，而且在阶段之间的通讯媒介采用的是"文档＋理解分析"，从而一方面带来了重复工作，另一方面阶段耦合也不是很紧密。

因此，软件工程的一个重要挑战是软件生产过程需要有一个根本性的改进。原先的重点主要放在生产过程的控制上，而不是放在改变生产过程从而达到生产的根本性改进之上，实现根本性的改进需要采取"构造性方法"。

我们认为软件系统的开发存在着其定义层(主要指软件开发前端的需求描述、分析及软件规格说明的形成阶段)与实现层(主要指软件系统的算法、代码、数据结构的实现与产品形成阶段)之间的断层。主要表现在：

①目前的生命周期开发方法把重点放在软件的设计实现层(具体算法、语言、数据结构的描述)，这样给系统的功能修改、更新维护与重设计工作带来较大的困难，尤其是软件文档与源码变得难以维护；

②对于用户的需求，经过需求分析阶段即细化成若干缺乏相关依赖性和一致语义的定义，从实质上脱离了就需求及功能定义与用户进一步反馈，定义不具有可操作性，而由人作解释后直接进入面向程序的实现层工作，形成明显的断层。

③软件设计中对灵活性、可修改性的要求越来越高，并且在软件修复、重用机制上也提出了新的要求。而断层的出现则影响了上述能力，也直接阻碍了软件自动生成。

为了解决这一断层问题，首先考虑：软件的开发问题实际是如何描述功能需求·引入一种形式化的定义方法，通过构造方法抽取软件系统的功能模型。软件的整个开发演变为基于定义的增量开发过程，即作为以后开发阶段的"唯一"基础，通过在此高层定义上的连续加工，直至底层产品的生成。这一种做法体现了三方面的优点：

应晶 博士生，主要研究领域为人工智能、CASE与KBSE。吴朝晖 博士，主要研究领域为人工智能、知识表达。何志均 教授，博士导师。

·软件产品的流水线作业；

·弥补了软件开发中存在的断层问题；

·给产品的维护、再实现等提供了可能。

本文指出了软件开发过程中新的方法论需求，并阐述了我们所提出的软件定义高层构造的方法论及其对软件开发过程的支持；在此基础上提出进一步的研究方向。

二、新的方法论需求

软件系统的开发至今已取得许多令人鼓舞的发展。正如计算机辅助设计为硬件设计带来革命性的飞跃，桌面印刷系统给原来的出版业带来革新一样，使人们感觉到软件自动化对软件设计带来的巨大效应正在酝酿和逼近。但事实表明，软件自动化的真正实现尚有一段漫长的道路，就象人工智能发展初期所提的雄伟目标一样似乎遥遥无期。在软件工程领域，如果不对软件开发理论和软件开发方法学进行细致的研究，建立不起正确有效的软件开发方法和技术，其他的一切（如工具、环境等的研究）都将无所依托。

改进软件生产过程的最常用的工具有程序度量工具和软件开发支持工具。前者有助于找到应用现有软件开发方法学时的瓶颈；后者有助于减轻生产过程中人的工作。但它们的角色还是辅助性的，并没有解决方法学本身更基本的问题。

另一方面，现有的各类成熟软件工具与技术已越来越成功地支持了各类软件系统的开发。这批从实践中产生的软件工具有其广泛的应用领域与背景，并辅助软件开发人员成功地解决了一些实际问题。虽然说它们的研究对软件工程学科的本质性突破没有直接的关系，但把研究工作建立在这些实用工具上是切合时尚的。而且也为软件自动化这一目标打下了坚实的基础，使人们意识到应该及时开展更为深入的研究。

按 R. T. Yeh[1]的观点，软件开发中存在"双重压迫"：一是大型系统的问题定义方法，二是复杂系统的维护问题。这两点造成了可怕的瓶颈。因此迫切需要引入一种新颖的方法论来指导软件系统开发，使软件生产率有巨大的改进，打破这一瓶颈。"为了避开陷阱，最好重新设计一条线路"。

因此，围绕软件系统开发过程中出现的断层，需要引入一种增量式定义开发方法论。

三、高层定义构造方法论

3.1 方法论的提出

本文提出一种支持软件定义高层构造的方法论（MHSC，Methodology for High-level Specification Construction）。MHSC 可描述为：从软件开发的需求定义层入手，研究软件定义的高层构造过程。提出一种支持多维语义的定义语言，并构造实际领域软件系统的合一化功能模型。在此基础上，通过变换、求精等方法的作用，逐步过渡到设计阶段的软件实现，以支持软件的自动开发过程。MHSC 所支持的是具有增生结构的软件构造过程。软件系统的开发不再是周期性的概念，而且需求分析、设计及维护等阶段之间的界限逐渐淡化。

此方法论强调的是一种多维语义的定义，统一地描述数据流、控制流、状态转换、实体关系及模块组织等语义信息（在以往的方法论中是分离的，而且每一维均有单独的处理工具，难以耦合），形成软件系统的一个可演化的单一模型（作为整个软件加工过程的核心）。

我们认为，对软件开发过程的描述分为七个阶段：获得完整的需求、描述上下文系统的交互、描绘（图解）子系统、组织成分（贯穿整个开发过程）、对象的动静态特性的定义、构造一个完整的设计模型、对此模型的加工过程，由此软件开发过程可简化为一个三元组：$SD=(I_i,O_i,P_i)$。其中：I_i：功能，资源，性能需求；O_i：提供了实现系统所需的定义；T_i：变换，求精，概念具体化及合成方法。

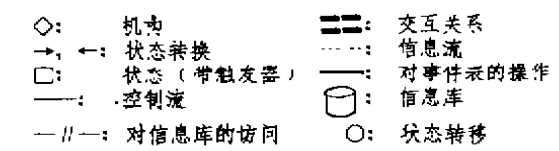
3.2 定义模型

我们强调可执行定义，即定义的可执行能力（关于定义的可执行性有一些争论，Fuchs[2]提出定义是可执行的，而 Hayes[3]则认为定义不一定是可执行的）。软件定义是一个概念模型同时又是一个行为模型，通过描述性语言可以支持定义的可执行性与表达能力两个方面，而且这使得抽象层上的程序验证变得可能。在后面提到的变换方法若与可执行定义相结合，则执行定义就形成了各阶段的文档。因此，可执行定义提供了一种直接过渡到系统设计的可能，Zave 认为它与实现之间的差别只在于资源的管理[4]。

我们对定义设计的出发点是：忽略需求定义和设计定义的界限，经典描述方法存在着缺陷；用 DFD 等不能描述大型系统中的"黑盒子"（指大粒度模块），所以必须涉及那些高层处理的底层细节才能反映系统的整体行为。我们的实现方法主要通过信息局部化、分离细节、表达可重用概念、定义并发动作的粒度、定义时序约束，并且希望语义的形成是基于概念模型而不是理论。这类语言要求能支持：

- 领域实体的描述；
- 各类变化(过程、处理、事件)；
- 约束、假设；
- 时间特性；
- 知识的组织；
- 抽象机制(求精)；
- 一致性检查、冗余消除；

在定义模型的构造中，支持下列的图示方法：



状态转移是定义构造的核心成分，定义为一个九元组：

Transition = (N, PreCond, PostCond, TC, GC, GEL, LEL, IF, A)，其中：

N：转移名称

PreCond：前条件(执行前须满足的条件)

PostCond：后条件(执行后须满足的条件)

TC：时序约束

GC：通用约束

GEL：感知的全局事件链(不同机构能访问的事件)

LEL：感知的局部事件链(仅用以激活本机构中的状态改变)

IF：状态转移过程的信息流

A：执行动作

因此，我们提出定义语言的描述模型 SDM：

$SDM = (\{St_i\}, \{E_i\}, \{A_i\}, \{Se_i\}, \{IbOp_i\}, \{ADT_i\}, \{AOT_i\}, \{P_i\})$

其中：$\{St_i\}$—状态集；$\{E_i\}$—事件集；$\{A_i\}$—组织集；$\{Se_i\}$—服务集；$\{IbOp_i\}$—标准信息库操作算子；$\{ADT_i\}$—抽象数据类型集；$\{AOT_i\}$—抽象操作类型集；$\{P_i\}$—谓词集

在系统需求分析过程中，定义的生成分为以下几个步骤：

—首先用户建立一个 ER 实体关系图(即系统的几个服务对象)；

—对每个实体，从系统状态集中选取合适的状态来构造子状态集(也可由用户重新创建状态)；

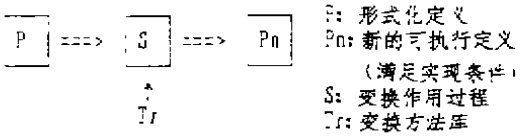
—单独构造几类信息库操作(从标准库中或自行构造)；

—构造状态间的转移。

3.3 变换方法

变换方法是指一个较为抽象的定义被重复地变换(细化)，变为更加具体的形式，直至一个目标系统的生成。C. Rich[5]曾预测：具有某种形式的变换法肯定是所有未来自动程序设计系统的组成部分。我们的工作以变换方法为基础，充分发挥变换过程的特色来体现软件开发过程的自动化程度。

MHSC 方法论能够很好地支持演化式的变换方法。变换并不是提供定义的实际实现，而是重新对定义进行求精。整个软件开发过程可视为不同定义层次上的变换序列。变换过程的体现如图所示：



SDM 作为一种单一的模型同时包括了定义和实现的构造，每个变换均是在同一种模型上作一些改变，要么用较底层的构造来替换高层构造，或是在同一层次上进行重组。SDM 实质上体现了使用变换来支持基于定义的原型开发的思想。

变换按不同的分类方法有多种，从变换的特征划分包括：扩展变换、收缩变换、限定(约束)变换、松驰变换、求精变换、抽象变换；从变换的作用分类：通用性变换(Berzins 在文[6]中作过一些分析)、特定领域中的变换(主要是我们在具体的实时服务领域提供的变换)、代码生成过程中的变换(程序变换)；从程序变换角度分：定义新的关系、定义一段代码、替换引用关系、从程序中移去、简化表达式、校验、定义新的数据结构。

我们引入两类变换，一类是特定领域的变换知识，针对领域的特征对合一化模型进行变换操作；另一类是面向目标语言的底层变换，用于代码层的程序生成变换。

(1)底层变换

—初始化变换

—出错处理变换

—程序渐进生成变换(模块生成过程)

—粗粒度模块调用关系变换

—粗粒度信息流传递变换

—上层模块扩展变换

—程序约束关系限定变换

—信息库操作变换

—辅助性模块变换

—公共模块筛选变换

—目标语言变换

—语言相关的全局特定变换

(2)领域变换

—如果服务类别是属于即时处理型的，如航空订票、饭店预订等；则系统中必须有预订模块，即必须提供资源的分配、释放算法。

—如果服务类别不属于即时处理型的，如旅行社、娱乐包场等；则系统中只需一个订单登记库即可，但须有一个额外的传输模块，负责与资源提供者的联络。

—如果一种服务的资源是周期性而非一次性的；则界面上应该包括定期的初始化模块(针对某一阶段的资源生成)。

—如果在模型定义中有查找功能描述；则底层服务中应建立资源库及资源分配库(订单)的标准查询模块。

—……

在基于变换的实现这一点上我们提出三个原则：

·定义描述模型必须是一种宽域语言，即能描述不同程度、不同层次的功能设计信息；

·为了获取转换规则，需要获取程序设计知识，基于这一点，需要研究目标语言的语言特性；

·支持手工的求精机制

在变换的作用模型上，我们结合人工智能技术来支持变换目标的形式化、变换的作用策略、变换选择的合理性及手工(专家)变换几个方面。把底层语言的生成视为问题空间，则变换的作用实际便是问题求解过程。通用机制从变换库中选取合适的变换作用到当前的问题空间中。通过对目标语言的语法及语义分析，可以概括成大量的知识作为求解策略，利用问题求解技术来自动地产生各个变换序列。对第二类变换采取了匹配策略，而第一类变换则通过多个步骤和作用规则来完成，通过定义变换方法的作用次序与优先级，同时支持用户的交互干预作用方式，从广义上讲，整个变换作用过程可视为一个软件产品。

3.4 求精方法

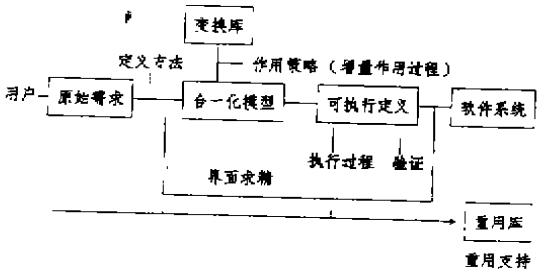
求精方法是对形成的定义模型的扩展。与定义描述标准相比较：控制流与状态描述已成形，直接支持定义的可执行过程；而信息流有待完善，模块组织需要进一步求精。变换是领域知识的综合体现，具有领域的适应性，而且包含了部分程序设计知识；求精主要是为用户的交互操作提供基本设施。因此把基于知识的支持体现在变换过程中；而把基于图形的支持体现在求精过程中。

提供图形设施支持定义的可视表达与各类可视操作，使得用户能在图示界面上对定义完成初始构造，并通过交互手段对定义进行求精，辅助整个变换过程。

在实现方式上，通过基于 icon 的可视技术，引入可视实体来实现上述的定义图示描述方法；提供各类可视操作(如整个构造模型的各类算子：更新、分割、分裂、组合、增加、提升、删除、低引、重新布局、嵌套、细化、抽象、聚集等)及丰富的交互手段对定义模型进行扩展和修改；同时支持模型的自动布局支持。

3.5 对开发过程的支持

整个 MHSC 方法论对软件开发过程的支持流程是：(见下图所示)



①分析某一类领域模型，并扩充或修改已有的重用库，更新各类变换及其作用策略，对各类机构、状态、事件、信息结构、服务、模块等加以访问并进行重新组织；

②利用定义描述模型构造软件的合一化模型(按照上述的步骤)，从重用库中匹配或抽取各种子模型及构造成分；

③由变换机制完成合一化模型的演化式变换过程，从模型变换到底层代码变换。同时对变换过程加以记录和跟踪，并提供变换过程的重现机制；

④支持可视集成界面的功能。在合一化模型的可视布局中应用各类求精算子、可视操作加以手工(或半自动)变换，同时支持模型的校正工作；

⑤体现定义的可执行能力。通过可视化仿真环境对定义模型进行解释执行，从总体上把握所构系统的功能；

⑥重用库的扩展，为同一领域的软件构造提供重用基础；

四、进一步的研究

1. 如何真正发挥定义的可执行能力是系统功能验证的关键环节，我们从下面几个方面考虑：

—利用图形方法对系统给予可视化；

—(从整体上)整个状态表的动态模拟；

—(从局部上)各种入口事件的连锁后继行为；

—对动作(ACTIONS)的模拟执行；

—定义执行时的可视化仿真环境。

定义的编译功能相对来说较难，所以可执行定义采取解释机制。可执行定义与可编译执行代码的差别在于系统控制机制的生成，如果采用解释方法，系统可以构造一个通用的系统控制机制(待定义检验正确后，封装成带控制的代码系统)。

2. 可执行定义能够有效地支持定义的证实与验证。其中定义的证实指用户需求描述与生成的定义在功能方面是否一致，定义的验证指可执行定义的能力与预想系统的功能是否符合。在这两方面我们只是做了一些探索性的工作，从实现角度作了简化处理，依靠在交互构造定义和反馈过程中来完成定义证实；通过在原型定义的执行过程中进行观察、在执行过程中的跟踪、记录、动态维护；执行过程中的不一致性检查、通过定义生成过程(求精与变换)的"重现"机制等来完成可执行定义的验证。从技术角度出发，验证和确认技术可以是手工的或自动的，简单的或用数学形式的。手工技术包括：复查、预排、检查等。使用数学的形式验证技术是以归纳断言、功能语义和显示语义为基础的。另一些数学形式化的技术是以静态分析、符号执行等为基础的。自动验证系统的例子有 Boyer-Moore 定理证明器开发的程序验证系统，标准验证程序，Gypsy 验证环境，以及爱丁堡的 LCF 系统。Yau 和 Wang 提出了验证分布式系统软件的通讯的一种方法。在这方面我们还有待于进一步的研究。

3. 对软件重用问题的考虑：人们能重用软件的唯一现实想法是要求软件的设计者一开始就考虑软件的重用性。我们认为应该把重用性作为初始设计的一个目标。在实际工作中，我们提供了能描述软件系统的定义模型、变换方法(包括变换序列)及各类构造成分的重用库，通过对系统标准库的访问、扩展来支持部分重用功能。事实上，通过软件定义高层构造的方法论来支持软件开发已经为以后的设计工作建立了重用基础。

4. 利用我们提出的软件定义的高层构造方法(整个构造方法的实现细节本文略)，我们成功地构造了一个实时航空订票系统。从开发过程中我们体会到得益最深的是软件系统功能原型的建立，而且连续构造过程从很大程度上保证了软件功能的正确性以及高效的更新机制。所有的开发工作均以合一化的定义模型为基础(利用所提供的机制比较方便构造出整个模型)。整个开发过程还是牵涉了一些手工操作，主要是底层变换后生成的代码的修改、扩充工作。因为我们的变换知识并不完备，有许多细节没有涉及，只能半自动地完成这个过程。另外一些事务性处理模块还须人工编程来完成。

5. 我们进一步的研究工作包括：

—变换方法及作用策略知识的完善。

—提供集成用户界面来维护整个动态构造过程(包括丰富的交互操作与可视设施)

—如何管理并有效地应用重用库

—下一步我们准备在已有基础上开发同类实时服务系统，从更深一层来验证 MHSC 方法论。

五、讨论

软件生产过程中存在的严重问题(大型化、复杂化而难以捉摸)影响了最后的软件产品。这个问题是由于难以改变软件生产过程，以致不能利用现代化技术造成的。即使有新的设备，新的工具，但常常感到难以改变既定的方法，难以把新的方法拼入到现有机构。所以说，新的设备与新的工具并不是根本之路，必须有新的方法论引入，打破软件的现有开发框架，而不能让目前手工劳动的局部优势(手工的问题同时又暴露了支持目前软件开发不力的问题)掩盖研究工作的意义。

本文的研究指出软件开发中存在的断层及其影响，基于这种现状与需求，文中指出，定义层是对用户所需系统的多维描述，能较为精确地把握，对整个开发过程起着不可估量的作用。我们所提出的 MHSC 方法论体现了"定义层—证实—可执行定义(快速原型)—验证—具体实现(实现层)—需求的反馈修改—定义层—重用—重实现过程"的流程。与传统的软件生命周期开发方法相比，MHSC 将以往软件工程生命周期中的发散性开发(多语义表达)统一到一致的多维语义的范畴中，使得软件开发更为聚焦，集中；改变了以往各开发阶段之间的跳跃性(不连续性；主要指中间软件产品的耦合度)，并在阶段描述及阶段特征、界限上有明显的区别。同时从软件生产过程控制角度，维护与测试已经不是两个单独考虑的环节，真正建立重用基础。为重设计工程及逆向工程开辟一个新的应用前景。 (下转第77页)

PC/CASE 系统使用了较好的 DFD 层次描述方法、由 DFD 自动生成结构图算法、完备的软件一致性检查，是一个支持软件开发全过程的 CASE 工具。该系统界面友好、工作台面向图形，能自动生成或保留中间软件文档，能自动生成 C 语言代码，为软件开发人员进行结构化程序设计提供了一个友好的开发环境。

参考文献

[1] T. G. Lewis, CASE: Computer-Aided Software Engineering, 1991, Van Nostrand Reinhold, New York

[2] Alan S. Fisher, CASE Using Software Development Tools, 1988, John Wiley & Sons, Inc.

[3] Poh-Tin Lee et al., Modelling of Visualised Data Flow Diagrams Using Petri Net Model, Software Engineering Journal, January 1992, p4-12

[4] Anthony Earl, Using a Raference Model for CASE Frameworks to Describe PCTE^{++}, SEE V.3, 1991, p123-142

[5] Richard Snpdgress, The Interface Description Language: Defition and Use, 1989, Computer Science Press

[6] R. J. Norman, M. Chen, Working together to Integrate CASE, 1992, IEEE Software

[7] C. Mcclure, CASE Is Software Automation, 1989, Prentice Hall. Englrwood Cliffs, New Jersey 07632

（上接第73页）

（1994年3月 MIT 的 R. Davis 在浙江大学讲学，提到知识表达的问题：较强的问题求解方法及合适的知识表达方法均不能彻底解决智能系统开发问题—主要由于现实世界和人们对问题的理解始终有一段距离（失真）。然而，我们认为软件系统的定义不尽然，因为人们对软件系统的功能需求是清楚的，只要能提供这样一种完善的领域模型定义语言（跟经验知识的描述不同），将为软件定义的实现提供自动化的可能。）

参考文献

[1] R. T. Yeh, 新型的软件演进风范，计算机科学，1991年第6期

[2] N. E. Fuchs, Specification are (preferably) exectable, Software Engineering Journal, Sep. 1992

[3] I. J. Hayes, Specifications are not (necessarily) executable, Software Engineering Journal, Nov. 1989

[4] P. Zave, Salient Features of an Executable Specification Language and Its Environment, IEEE SE-12, 2(1986)

[5] C. Rich, et al., The Programmer's Apprentice: A Research Overview, IEEE Computer 1988, 21(11): 10-25

[6] V. Berzins, et al., Using Transformations in Specification-Based Prototyping, IEEE SE-19, 5(1993)

[7] Michael R. Lowry, Software Engineering in Twenty-First, AI Magazine, 1992, Vol 14, №.3（亦见本刊94№2）

[8] Michael R. Lowry, R. D. McCartney, Automating Software Design, AAAI Press, 1991

[9] D. R. Smith, et al., Research on Knowledge-based Software Environments at Kestrel Institute, IEEE Tran. on Software Engineering, 1985, 11(11): 1278-1295

[10] D. R. Barstow, A Perspective on Automatic Programming, Readings in Artificial Intelligence and Software Engineering, 1986, Eds. C. Rich and R. C. Waters, 537-559, San Mateo, Calif: Morgan Kaufmann

[11] R. Balzer, A 15 Year Perspective on Automatic Programming, IEEE SE-11, 11(1985)

[12] V. Berzins, Luqi, An Introduction to the Specification Language Spec, IEEE Software, Mar. 1990

[13] P. Zave, An Operational Approach to Requirements Specification for Embedded Systems, IEEE SE-8, 3(1982)

[14] Y. Tung, et al., Multiple Views of an Executable Software Specification Language, J. Systems Software, 21(1993)