支持软件开发的可执行定义方法*

应 晶 何志均 吴朝晖

(浙江大学人工智能研究所 杭州 310027)

摘要 本文分析了软件开发过程中的可执行定义方法及其与软件开发的关系,并将作者提出的可执行定义方法论 MHSC(methodology for high-level specification construction)与现有方法论进行比较. 论文表明可执行定义方法将在软件系统从定义开发到系统实现的进化式开发过程中起着关键的作用,有积极的研究意义.

关键词 可执行定义,软件方法论.

中图分类号 TP311

1 软件定义的角色

软件定义是目标系统在一个概念层次上的精确描述. 开发人员和用户的需求通过定义阶段得到反馈. 在理想的情况下,形式化定义反映了开发人员及用户对于待实现系统的统一认识,体现出系统的精确性和完整性. 但是对精确性的要求往往使得定义难以开发. 因此,在分析阶段建立和形式化的需求往往是不清楚和不完整的. 另一方面,开发者要在用户需求的解释基础上作出决定,如果应用领域没有较好地理解,需求往往被曲解. 在软件开发的早期阶段,开发人员需要对未来的软件产品作出关键的决策,用户亦需要证实这些决定来保证初始需求得到满足. 然而,开发人员与用户有着不同的背景,使用不同的术语,这种交流的障碍是软件开发的一个严重问题. 因而,软件往往到了可执行状态下才能真正得到证实.[1]

当前软件工具有着广泛的功能,但它们的一个共同弱点是缺乏模型的严格语义;定义不能加以执行,不可能从定义中自动生成代码. 许多原型辅助系统实际上是一个围绕高层语言的工具集和一个数据库管理系统. 尽管能缩短开发时间,但它们没有软件中间产品的导出能力,因而难以求精用户的需求. 对软件定义的支持应满足:定义构造的高度交互性和灵活性;需求的增量提交(从非形式化到形式化);定义的自动处理(可检查和可证实);并且允许定义是不详尽、不完备,甚至是不一致的. 通常采用的一个标准是在整个项目中对定义的增量开发提供主动支持,同时支持可执行定义方法.

---

* 本文研究得到国家自然科学基金和国家 863 高科技项目基金资助. 作者应晶,1971 年生,博士,副教授,主要研究领域为人工智能,基于知识的软件工程,CASE. 何志均,1923 年,教授,博士导师,主要研究领域为人工智能,智能软件开发环境,CAD. 吴朝晖,1966 年生,博士,副教授,主要研究领域为人工智能,软件开发环境.

本文通讯联系人:应晶,杭州 310027,浙江大学人工智能研究所

本文 1996-05-06 收到修改稿

© 中国科学院软件研究所 http://www.jos.org.cn

2 可执行软件定义

精确性与可理解性的冲突引出可执行软件定义的研究.[1,2]可执行定义表示了一个能产生目标系统与之环境交互行为的系统模型.在理想情况下,产生的行为非常接近所定义软件系统的期望行为.进而,可执行定义不仅具有开发人员需要的精确性和完整性,还使用户能通过运行定义来迅速证实早期的决定.可执行定义的研究主要集中在合适的定义语言上.[3~5]代数定义,功能型程序设计和逻辑程序设计的提出均作为可执行定义的基础.其核心思想是把数学基础与过程性语义相结合.数学基础支持抽象而且与运行无关的定义,并对这些定义作形式化的推理.过程性语义允许通过使用定义语言的解释器来执行定义.然而,一种描述性的定义语言并不能形成一个成功的可执行定义方法论.论文认为在目前的可执行定义的研究中忽略了以下几点而未加实现:

① 可执行定义没有显示出一种实际的行为.相反,仅是指出软件系统的局部功能.

② 一旦完整的行为(包括用户界面)可以定义,则必须定义面向实现的细节.这样使得细节问题难以从面向问题的定义中分离出来.

③ 进一步,可执行定义可以从上至下地开发和证实,而不是自底向上.因此定义的整个行为在所有细节指明以前不能加以证实.

④ 再者,一些用于可执行定义的语言对用户来说难以理解.这种障碍并不影响通过执行的证实,而是常常使得用户不能主动参与开发过程.

文献[6]提出的MHSC方法论着重考虑:

① 通过面向实际领域的仿真与模拟功能,引入资源的操作与分配,不仅体现系统所具有的各项功能,还从整个实际运行环境中观察软件系统的整体行为;

② 通过软件定义和用户界面的分离,把可执行软件原型置入另行构造的环境界面中,通过界面反映软件系统的行为特性,而对软件定义的更新则与具体界面无关,大大增强定义的维护能力;

③ 支持多粒度的功能证实,从系统的外部行为、模块之间的交互、具体操作过程直至语句的执行;

④ 合一化模型体现的多重视图和强交互机制减轻了用户的理解难度.

文献[7]提出定义是可执行的,文献[8]则提出定义不应该是可执行的,因为前者的观点限制了定义语言的表达能力(认为不可执行的定义可以在几乎是同一个抽象层上变成可执行而不改变它们的结构).作者认为,软件定义是一个概念模型又是一个行为模型,通过可执行性使得在抽象层上的验证变得可能.如果把可执行定义结合变换方法,则执行定义形成各阶段文档.另一方面软件定义的描述还包括非功能性需求(度量、调度、管理).

可执行定义的使用也可以理解为原型.原型作为一种可执行定义加以构造,许多结果都与可执行定义有关.但原型方法一般侧重于系统的某些方面,如体系结构、用户界面、强调了原型开发的自动支持,通常不关心原型之上的抽象与形式化推理,这是差别所在.解决软件问题的实质性困难,而不是偶发性困难的一项工作是开发支持软件定义设计所需的方法和工具.[9]

© 中国科学院软件研究所 http://www.jos.org.cn

综上,定义的执行包含对需求的证实及定义的部分功能验证.从实质而言,可执行定义继承了形式化定义的特点[10]:问题的细节显式化、定义语言具有较好定义的语义、许多任务可以自动化.

开发可执行定义方法论的主要挑战在面向实现与面向问题的细节分离、可理解性的提高、实践行为的产生、避免受限的精确表达.面向实现的细节对于使定义变得可执行而言是必要的,如果它们没能从面向问题的定义中加以分离,则可能得到的是一个设计而非定义.[5,8]同样地,假设和异常也应从正常状态下的定义中分离出来.对开发人员来说,形式化问题难以解决.可执行定义易于在许多细节还未知或不确定时在早期阶段便形成受限的表达.

可执行定义的特点体现在允许一个软件系统在真正实现之前表现出实际行为能力,尤其是逻辑定义语言,结合了高度表达能力和可执行能力,支持在所要求抽象层上的面向特性和面向模型的定义描述.同时可执行定义是构造性的,不仅针对解的存在性,而且实在地加以构造,并不限制可能实现的选择.在此基础上采用变换方法有可能减小定义实现验证的必要性.

3 一个可执行定义方法论的设计标准

一个定义方法论由3个部分决定:需求描述模型、定义语言和定义开发方法.此外一个特定定义方法论的作用从本质上受到已有工具提供的自动定义支持的影响.可以建立一个新的可执行定义方法论的设计标准集.

文献[11]强调首先定义一个需求描述模型的重要性.此模型能建立理解应用领域世界和讨论系统输出的基础.系统和环境特征的定义把定义和现实连接在一起:定义的对象/事务个体反映了现实,而定义的结构与应用领域的结构建立了关系.描述模型作为一个用户能感知的系统概念模型.从这些原因出发,研究者认为模型必须具有知识表达模型中的表达能力.[12]

定义语言应用于预见的问题领域并且足以描述具有较小复杂性的非试验系统行为.因此定义语言或者是通用的(适于若干领域),或者加以调整后可应用到不同领域.定义语言是可增量式的,能容忍不完整性并且易于理解.再者,定义在正常情况下能清晰地区分正常与错误行为以及异常或可选择的行为.要求一种数学理论来支持语言和方法学,而且理论表达能包括描述性的语义,不管其形式化、语言的语法是灵活的.异常条件的定义和处理亦是自然和精确的.对于可修改性、可重用性和可读性等特点,使得可执行定义有可能分解成模块化对象.理想地,定义语言还支持类型抽象化、参数化和继承性等技术.最后,语言应适于变换以产生不同的表达.

与定义语言一样,定义开发方法除了用于软件开发范畴外还应适于应用领域.开发方法允许一种系统化的方法来形式化需求并支持迭代和高度交互的开发过程.特别注意考虑增量式开发,包括不完备定义或定义组成部分的证实.定义开发支持包含用于综合与分析定义的集成工具集.这些工具存储定义片断,完成可执行配置,提供不同的表达与视图,并管理版本与重用库.在必要时候,应提供用户界面和特定资源(数据库系统)及没有显式指明的组成

© 中国科学院软件研究所 http://www.jos.org.cn

的自动仿真过程.针对定义不可避免的形式化特征,采用不同的简单表达及图形设施来提高开发过程的可视能力尤其显得重要.

论文提出的以下几个方面是一组亦可应用于非执行定义及其相关方法论的分类标准.由于标准涉及定义方法论中不止一个部分,因此它们不针对这些组成部分来分类.如描述模型和定义语言在多数情况下均紧密相连,定义开发方法和可用工具总是相互影响.这些标准针对技术和方法特性,而不是组织或管理问题.这些分类标准是:形式化基础(理论)、范围(需求类型)、形式化层次(表达)、专门化程度(问题类别)、阶段(组成的类别、对软件开发过程的阶段适应性)、开发方法(构造)以及工具.

形式化基础:这一标准涉及描述模型和定义语言所包含的理论、描述成分的语法和语义.形式化基础决定了系统的抽象层次、自动化开发本质、验证和变换以及定义的模块化设施.描述方法有:逻辑、谓词演算、逻辑程序设计、应用程序设计、功能型程序设计、代数定义、状态转移模型(有限状态机)、数据抽象、过程抽象.

范围:描述模型、定义语言以及需求的可能类型的范围决定了它们各自的描述能力.描述方法有:功能性需求和非功能性需求.功能性需求对系统组成和环境的行为建模.非功能性需求主要指界面、性能(时间和空间资源界限、安全性、可修复性)、操作约束(可靠性、容错等)、生命周期和经济约束.非功能性需求象性能和操作性需求在实时(分布)系统的定义中尤为重要.非功能性需求的定义具有特殊的意义,它们难以形式化并广泛地集成到描述模型之中.

形式化层次:形式化层次直接影响了定义的可分析能力与测试能力,包括可能的自动建档与变换.描述方法有:非形式化、格式化、半形式化和形式化.显而易见,可执行定义通常是形式化的,但有着象数据类型约束上的差别.此外,具有普遍性的是功能性需求采用形式化叙述,同时非形式化需求仅仅非形式化地描述,如注释.如果形式性和完整性能以灵活的方式处理,不完整的可执行定义亦能处理.其它方法有:解释的、编译的、变换的;强类型;语言表示法、图形表示法、图形表达的自动生成;适于非完整可执行定义方法.

专门化程度:对特定领域的描述模型专门化增加了可分析能力,但减轻了灵活性和应用能力.可显式表达的系统相关方面与可能的应用域需加以考虑.描述方法有:领域无关、领域相关、专门领域;数据、控制、异常.

阶段:定义开发方法可特定对应到定义阶段,但它亦可扩展至设计阶段或后续变换阶段.描述模型和定义语言可以限于证实与验证中的使用.描述方法有:软件定义、系统设计、变换;定义验证、定义证实、系统验证.

开发方法:使用的合适方法.描述信息有:传统的、进化、变换的、基于知识的开发、对较难定义的问题的开发概念.其他特性有:人机界面类型、自动化程度、复杂度控制.对定义开发方法的描述有:功能分解、面向对象的定义、数据驱动的设计.进一步的方面还有概念化、验证与证实.

工具:工具能支持定义构造,通过自动化某些过程来辅助定义验证;通过抽取解释和模拟指定系统行为来辅助证实定义,利用图形来增强定义理解.描述方法有:集成编辑器、图形编辑器、基于知识的助手;解释器、编译器;自动建档、图形展示;预定义的定义组成.

© 中国科学院软件研究所 http://www.jos.org.cn

4 可执行定义与软件开发

可执行定义与非执行定义的重要差别是描述的系统行为可以在软件开发早期进行观察和证实. 作者用图 1 描述.

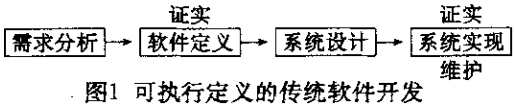


图1 可执行定义的传统软件开发

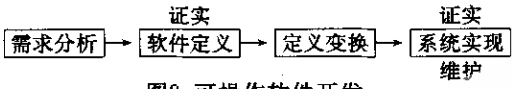


图2 可操作软件开发

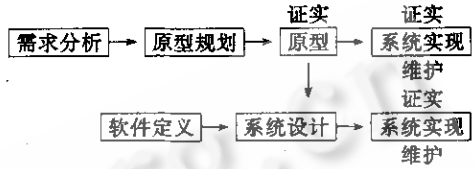


图3 面向原型的软件开发

可执行定义进一步可用于不同的软件开发范畴. 为了排除涉后的证实工作与实现层上的集中维护、操作性开发、面向原型的开发及变换式实现等方法逐步形成. 对于操作性软件开发范畴,如图 2 所示.

面向应用的变换阶段的目的是在改变行为生成机制时保持原有的外部行为. 而操作性范例既没有一个显式的设计阶段,也没有通过连续求精开发的实现过程. 它是一种进化式方法:软件系统从定义中逐渐演变进化而成.[13]维护工作在实现级上完成.

原型是建造目标系统的工作模型过程. 通常用程序设计语言编写的原型可能在功能和规模上受限. 其目的是阐明系统的特点与操作并减轻开发人员与用户之间的通讯. 至此,系统的外部行为非常重要,而如何实现这种行为则显得次要得多. 如图 3 所示.

变换式实现,亦称自动程序设计,意味着把定义自动地变换成目标系统.[3,4]与操作性软件相反,这一范例强调了应用变换的自动工具的扩展性支持. 这些变换不仅形成了系统实现,还以(半)形式的方式对变换与决策加以记录. 如图 4 所示.

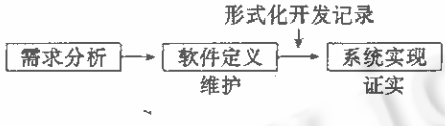


图4 变换式实现

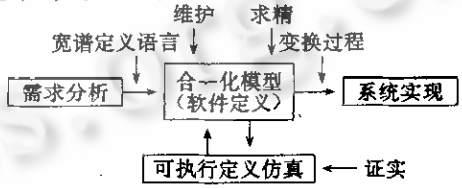


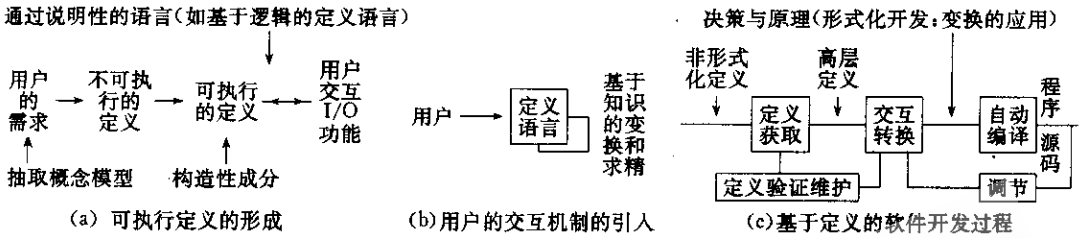
图5 基于高层构造的软件开发

变换的具体例子有表达的变化、算法的选择、优化和编译等. 这些变换决策或交互生成或通过自动辅助,如用专家系统加以实现. 如果一个软件由变换式开发方法来完成,则仅需维护其形式化定义(正如对一个软件系统修改维护时只需修改源程序,然后直接编译即可).

MHSC 方法论体现的是一种支持软件定义高层构造方法,不同于以上 4 种开发范例(图 5 所示). MHSC 通过支持软件定义增量变换并有机结合形式化变换方法(基于变换知识的推理过程)来开发目标系统. 由于定义的不完整以及形式化的变换知识作用过程的不完备性,作者强调采用求精方法支持软件定义的交互式变换过程. 通过对高层合一化模型所展现的可执行软件定义进行仿真,支持定义的证实和系统功能验证.

作者强调可执行定义,其形成与构造过程如图 6 所示. 可执行定义到实现的过渡只差于资源的管理,因而是一种从需求到实现的巧妙过渡方式,并引进了软件构造方式.

© 中国科学院软件研究所 http://www.jos.org.cn



(a) 可执行定义的形成　(b)用户的交互机制的引入　(c)基于定义的软件开发过程

图6

5 MHSC 方法论

5.1 方法论的提出

文献[6,14]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):从软件开发的需求分析与定义层入手,提出一种能支持多重语义维描述(包括数据流、控制流、实体关系、模块组织、状态转换等)的软件定义语言,应用语言来构造实际领域软件系统合一化功能模型,作为软件需求分析中间结果,在此基础上通过进化方法(基于知识的变换方法、基于可视技术的求精、基于模拟机制的软件定义证实与系统功能验证),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,从本质上改进现有的软件生产过程. MHSC 所支持的是具有增生结构的软件增量式构造过程.

MHSC 提供了 3 个方面的支持:

① 定义语言的构造:提出一种描述不同粒度、不同层次的功能设计信息的定义语言. 针对面向的反应式领域的系统描述具有充分的表达能力,能形成统一地包括多语义维的目标系统的合一化功能模型作为开发的中间产品,通过对模型的进化方法逐渐生成实现层的系统构成(模块化、程序化及可执行性). 支持软件开发的多重视图表达与展现形式;、

② 基于知识的变换方法:在领域分析的基础上形成大量的定义变换知识(规则). 结合作者对专家系统的研究基础[15],实现一个支持定义变换进化过程的开发框架,体现基于知识的表达方法与问题求解能力;通过应用自动变换与手工变换的过程,反映领域变换方法对软件实现的支持特点;

③ 可执行原型的形成:利用定义语言的可执行能力,在变换与求精过程后形成一个可执行的软件原型,从外部行为反映系统的整体功能. 在此基础上,提供定义仿真环境,支持定义的证实与验证,把实现层的系统组成与定义层的用户需求建立一个反馈,使目标系统能满足开发初期的用户实际要求.

5.2 定义变换与求精

MHSC 方法论以变换方法为基础,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,并能够较好地支持演化式的变换方法. 软件的整个开发过程视为不同定义层次上的变换序列. 其过程体现如图 7 所示.

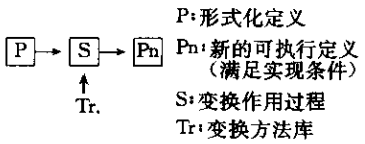


图7

基于变换的实现上作者提出 3 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并着重研究目标语言的语言特性;支持手工的求精机制. 因此,在变换的作用模型上,作者结合人工智能

© 中国科学院软件研究所 http://www.jos.org.cn

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变换[14,15]，利用基于知识的推理技术自动产生各个变换序列，支持人工干预对变换的选择和作用. 从广义上讲，整个变换作用过程可视为一个软件产品.

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；求精是对形成的定义模型的扩展，主要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；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，使用户能在图示界面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.

5.3 支持模式

MHSC 方法论对软件开发过程的支持模式如图 8 所示.

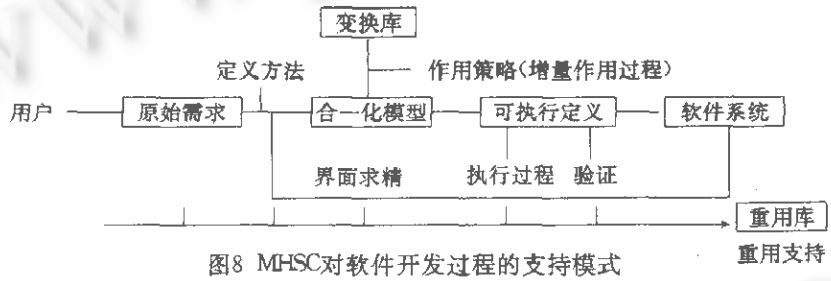


图8 MHSC对软件开发过程的支持模式

其开发流程为：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，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.

5.4 方法论比较

作者通过对现有的可执行定义方法论进行分析[7,16~21]，并与 MHSC 加以比较：

形式化基础：大致上基于逻辑、功能型程序设计和基于代数定义的系统数目均等. 在许多情况下，一个系统还基于状态转移模型. MHSC 则采用功能型程设结合状态转移图方法.

范围：所有系统允许指明功能性需求. 一些系统允许定义时序约束，而用户界面定义不是未提及便是显式地排除在外. MHSC 不仅提供了功能性需求，还对实际环境中的全局/局

© 中国科学院软件研究所 http://www.jos.org.cn

部约束,包括资源分配、使用均给出了定义.

形式化层次:某些解释型定义语言可以运行不完整的定义.大多数系统提供了语言方法.仅有少数系统提供了图形表达,还有一些允许生成图形展开.MHSC 支持语言定义方法,同时提供了多语义维的图形描述方法,并提供多重视图的展现功能.

专门化程度:多数系统是领域无关并允许定义数据与控制,少部分支持异常的定义.MHSC 支持的软件定义包括领域相关(实时反应式领域)的数据、控制及扩充的异常处理(多种入口包括定时器、触发器、手工操作以及各种异常出口的处理).

阶段:所有系统主要用以定义开发;部分系统还考虑了定义的验证.此外,一小部分系统可应用于系统设计或适于变换.MHSC 支持软件定义的增量式变换,定义的证实及基于模型可执行能力的定义验证.

开发方法:一些系统集中于传统软件开发,多数用于进化式软件开发.半数系统还提供了一种定义开发方法.2 个系统应用了面向对象的开发.MHSC 支持一种结合进化式软件开发与变换式实现的基于高层定义构造的软件开发方法.

工具:除了解释器和编译器,一些系统提供了分析工具,一个专门的图形编辑器,或视图生成器.少数系统提供了基于知识的工具.其中有一个系统还在定义组成中显式地包含了一个数据库.MHSC 支持视图生成器、图形编辑器、基于知识的变换作用工具、视图展示功能等可视设施.

通过下表可以反映 MHSC 在可执行定义方法论标准模式中所处的角色.

① 形式化基础

逻辑	谓词演算	逻辑程序设计	应用程序设计	功能型程序设计
			PS	WS

代数定义	状态转移模型	数据抽象过程抽象
	WS	WS

② 范围

	功能性需求		非功能性需求			
系统组成功能	行为建模	界面	性能	资源	操作约束	生命周期	经济约束
WS	WS			PS	WS		

③ 形式化层次

非形式化	格式化	半形式化	解释	编译	变换	语言表达	图形表示	自动生成
		WS			WS	WS	WS	WS

④ 阶段

软件定义	系统设计	变换	定义验证	定义证实	系统验证
WS	PS	WS	PS	PS	

⑤ 开发方法

传统	进化	变换	基于知识的开发	人机界面	自动化程度	复杂性控制
	WS	WS	PS		PS	

功能分解	面向对象的定义	数据驱动的设计	概念化	证实	验证
WS	PS		PS	PS	PS

⑥ 工具

集成界面器	图形编辑器	基于知识的助手	解释器	编译器	自动建档	图形展示
WS	WS	PS	PS		PS	WS

(其中 PS:部分支持, WS:较好支持, 其它:没有支持)

© 中国科学院软件研究所 http://www.jos.org.cn

6 对软件开发的影响

可执行定义方法可以大大减少软件开发过程中的回溯工作量，与此紧密相关的是系统的合适性问题(“此系统真正是我们所需要的系统吗?”). 目前的方法包含通过建立结构模型来进行原型设计和模拟系统行为. 原型设计的主要想法是在开发过程中“...为探索有可能实现什么提供了指导...”. 这意味着，在开发期间产生一系列更详细、更完善的模型(规格说明)和原型(实现方案). 希望软件定义与原型两者的共存和共生能以一种有利的方式相互促进. 通过原型构造所获得的经验和见识将统一对实际需求的见解并发现错误或改进的途径.

下一阶段的软件开发将通过从开发周期各阶段的需求推出的一个互相协调的语言系列来支持开发过程. 因此，生成软件的过程将是一个受综合定义系统支持的定义、变换和确认的连续过程. 在一种语言中定义和确认目标系统后，就进展到下一阶段，并确立各种描述形式之间的明显关系. 整个开发过程的全部信息将以一种灵活的内部格式存储在中心数据库. 这个软件开发数据库将是各种语言和合成工具之间信息存储与交换的中心. 定义语言相对局限于软件工程的一个特定阶段(如 RSL，DSL，PDL，编程语言). 可以证明，由于当前语言类型的增广，这些阶段间的差别将变得不明显. 此外，定义语言的特征将发生变化，产生混合形式/非形式与图形/文本表示等方式，因此必将扩展定义语言的维数.

软件落后于硬件，而且这种落后只能靠提高生产率来扭转. 软件可再用性和自动编程为达此目标提供了最有希望的途径. 而定义语言却是实现软件可再用性和自动编程的前提.

7 结 论

本文着重讨论了可执行定义方法论特征. 软件定义在现有软件开发过程中起着重要的角色，它作为系统设计的工作基础直接影响着软件产品的质量. 软件的大量开发实践反映了一种新的需求，即是可执行的定义软件. 这中间有其特点与产生的动机，及实际研究中面临的挑战. 可执行定义作为软件进化式开发过程中的关键思想，为软件的新一代开发范例即基于高层定义构造的软件开发建立了基础. 通过对现有的可执行定义方法论分析，依据分类模式下的评价标准，总结了现有系统的共性和个性特点，并着重分析了 MHSC 方法论与现有方法论的不同点及其对软件开发的影响，论文所讨论的可执行定义方法将有可能为新一代软件开发建立基础.

致谢 陈火旺教授和孙永强教授对 MHSC 方法论提出许多有益的建议，在此表示感谢.

参考文献

1 Agresti W W Ed. New paradigms for software development. IEEE Computer Society Press, Washington, 1986.
2 Zave P. The operational versus the conventional approach to software development. Communication of ACM, 1994, **27(2)**: 108～119.
3 Balzer R *et al*. Operational specification as the basis for rapid prototyping. ACM Software Engineering Notes, Dec. 1982, **7(5)**: 98～108.
4 Barstow D R. Rapid prototyping, automatic programming, and experimental sciences. ACM Software Engineering Notes, Dec, 1982, **7(5)**: 109～120.
5 Zave P. An operational approach to requirements specification for embedded systems. IEEE Trans. SE-8, 1982, **3**:

© 中国科学院软件研究所 http://www.jos.org.cn

250～269.
6 Ying Jing, He Zhijun, Wu Zhaohui *et al*. A methodology for high-level software specification construction. ACM Software Engineering Notes, April 1995, **20(2)**: 48～54.
7 Fuche N E. Specification are (preferably) executable. Software Engineering Journal, Sep. 1992, **7(5)**: 323～334.
8 Hayes I J. Specification are not (necessarily) executable, Software Engineering Journal, Nov. 1989, **4(6)**: 330～338.
9 Brooks F. 软件工程之实质性困难与偶发性困难. 计算机科学, 1988, **3**: 43～52.
10 Gehani N. Specifications: formal and informal—a case study. Software Practice and Experience, 1992, **22(12)**: 827～838.
11 Cameron J R. An overview of JSD. IEEE Trans., SE－12, 1986, **2**: 222～240.
12 Borgida A *et al*. Knowledge representation as the basis for requirement specifications. IEEE Computer, Apr. 1985.
13 Puncello P P *et al*. ASPIS: a knowledge-based CASE environment. IEEE Software, Mar. 1988, **5(2)**: 58～65.
14 应晶. 一种新的软件系统开发方法论研究[博士论文]. 浙江大学, 1995.
15 Wu Zhaohui, Ying Jing, He Zhijun. A novel toolkit to build coupled expert systems. Science in China (series A), 1995, **24(suppl.)**: 73～80.
16 Zave P. Salient features of an executable specification language and its environment. IEEE Trans. SE－12, 1986, **2**: 312～325.
17 Ghezzi C *et al*. TRIO: a logic language for executable specfications of real-time system, J. Systems Software, 1990, **12(2)**: 107～123.
18 Tse T H, Pong L. An examination of requirements specification language. The Computer Journal, 1991, **34(2)**: 143～152.
19 Berzins V, Luqi, An introduction to the specification language spec. IEEE Software, Mar. 1990, **7(2)**: 74～84.
20 Wing J M. A study of 12 specifications of the library problem. IEEE Software, July 1988, **5(4)**: 66～76.
21 Tsai J J P. HCLIE: a logic-based requirement language for new software engineering paradigm. Software Engineering Journal, July 1991, **6(4)**: 137～151.
22 何志均, 应晶等. 一种基于 Petri 网的软件定义构造方法. 软件学报, 863 专刊, 1996. 273～278.

BUILDING EXECUTABLE SPECIFICATION TO SUPPORT SOFTWARE DEVELOPMENT

YING Jing　HE Zhijun　WU Zhaohui

(*Artificial Intelligence Institute　Zhejiang University　Hangzhou*　310027)

Abstract　This paper analyzes the executable specification building methods in software development procedure and its relationship with software development, then makes a comparison between MHSC (methodology for high-level specification construction) and currently existed relevant methodologies. The MHSC methodology put forward by the authors supports executable specification explicitly. The paper declares that executable specification building method will play an important role in the evolutionary process from software specification to system implementation. Its research will turn out to be of great significance.

Key words　Executable specification, software methodology.

Class number　TP311

© 中国科学院软件研究所 http://www.jos.org.cn